UNIVERSITA' DEGLI STUDI
DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

Honeywell Bull
CENTRO STUDI
INFORMATICA
E AUTOMAZIONE

NOTE DI SOFTWARE

ANNO 1987
38
DICEMBRE

Note di software è una pubblicazione trimestrale, edita a cura del Dipartimento di Scienze dell'Informazione dell'Università di Milano e dal Centro Studi Informatica e Automazione della Honeywell Bull Italia.

Note di software intende costituire uno strumento per la circolazione di informazioni, idee, metodologie ed esperienze nell'area delle tecnologie e della produzione del software.

La collaborazione a **Note di software** è libera. Chi desiderasse pubblicare un contributo è pregato di prendere contatto con il Direttore responsabile o con il Comitato di Redazione. In particolare si sollecitano contributi nella forma di monografie, da inviare in triplice copia, e che non dovrebbero superare le cinquemila parole.
I lavori ricevuti per la pubblicazione verranno revisionati dai membri del Comitato di Revisione Scientifica, che possono avvalersi della collaborazione di specialisti del settore. I lavori pubblicati riflettono comunque le opinioni dei loro autori.

Note di software viene distribuito ad Aziende, Enti e specialisti del settore che ne facciano richiesta alla Redazione.

Direttore Responsabile: Paolo Ghelfi
Honeywell Bull Italia S.p.A., via ai Laboratori, Pregnana Milanese

Comitato di Redazione: Stefania Bandini, Francesco Gardin, Roberto Polillo
Università di Milano
Dipartimento di Scienze dell'Informazione
via Moretto da Brescia, 9 - 20133 Milano
tel. (02) 2772233

Comitato di Revisione Scientifica: Alberto Bertoni, Gianluigi Castelli, Giovanni Degli Antoni, Giorgio De Michelis, Mario Italiani, Gaetano Aurelio Lanzarone, Giancarlo Martella, Marco Maiocchi, Giancarlo Mauri.

Autorizzazione del Tribunale di

ARCHIVIO STORICO
ASSOCIAZIONE POZZO DI MIELE
FDPM-89 RNSW126

UNIVERSITA' DEGLI STUDI
DI MILANO
DIPARTIMENTO DI SCIENZE
DELL'INFORMAZIONE

FPDM - 89

Honeywell Bull
CENTRO STUDI
INFORMATICA
E AUTOMAZIONE

QUESTO NUMERO 1

— ARCHITETTURE E ALGORITMI SISTOLICI NEL CONTESTO DEL CALCOLO PARALLELO
di S. Bandini, M. Goldwurm, G. Mauri 3

— ARCHITETTURE SISTOLICHE
di R. Grande 11

— UN DRIVER PER SISTOLICI
di G. Mazzocchi, S. Missaglia, D. Mella, F. Berno 23

— RAY TRACING E ARCHITETTURE SISTOLICHE
di S. Zanantoni 32

— A THEOREM PROVING REVIEW FOR PROOF SYSTEM BASED PROGRAMMING
di M.P. Bonacina 40

— QUADTREE PER LA MEMORIZZAZIONE DEI DOCUMENTI IN FORMATO IMMAGINE SU ELABORATORI PARALLELI
di C. Anedda, L. Felician 45

AVVENIMENTI 54

— La Logique dans l'Informatique 54

— LOGIC COLLOQUIUM '88 55

NOTE DI SOFTWARE

ANNO 1988
38

QUESTO NUMERO

Questo numero di Note di Software contiene alcuni articoli dedicati alle architetture ed agli array sistolici, che raccolgono parte del materiale presentato in un corso tenutosi presso la FAST/FOIST nel 1987.

Gli array sistolici, introdotti recentemente, rappresentano una particolare forma di sistema multiprocessore parallelo, e sono costituiti da più processori identici collegati secondo strutture geometrico-topologiche regolari. I dati vengono passati da un processore a quelli confinanti con un flusso costante e regolare, e interagiscono all'interno dei processori in cui si incontrano.

L'uso di architetture sistoliche consente di realizzare algoritmi molto veloci per numerosi problemi, tra cui hanno particolare rilievo anche applcativo quelli relativi all'elaborazione di segnali e di immagini ed al riconoscimento della voce, oltre a classici problemi di calcolo numerico come il prodotto di matrici, la convoluzione, la trasformata di Fourier veloce.

L'articolo di apertura, di S. Bandini, M. Goldwurm e G. Mauri, presenta una breve rassegna delle problematiche principali relative al calcolo parallelo, collocando le architetture sistoliche in questo contesto. L'articolo di R. Grande descrive più in dettaglio le architetture sistoliche, presentando come esempio algoritmi relativi ad alcuni dei problemi che sono stati con maggior successo trattati con queste tecniche. L'articolo di G. Mazzocchi, S. Missaglia, D. Mella e F. Berno presenta un sistema di sviluppo basato sul sistolico programmabile GAPP (Geometric Arithmetic Parallel Processor) NCR, ed il relativo assembler, descrivendo in dettaglio uno strumento software, sviluppato presso il Laboratorio del Dipartimento di Scienze dell'informazione di Milano, che ne ha permesso l'integrazione con il tradizionale mondo MS-DOS. Infine, S. Zanatoni descrive un algoritmo sistolico per il calcolo della intersezione tra raggio ed oggetto nel metodo di ray tracing per la simulazione visiva, e ne presenta una implementazione sulla scheda GAPP NCR. Le presentazioni del programma sistolico vengono valutate in termini quantitativi e confrontate con le presentazioni di due programmi sequenziali in C per lo stesso problema.

Gli altri due articoli che concludono il numero sono firmati, in ordine, da M.P. Bonacina e da L. Felician.
Il primo offre una rassegna di facile consultazione sulla dimostrazione automatica di teoremi, mentre il secondo riguarda i problemi della memorizzazione di testi e documenti mediante quadtree e la possibilità di utilizzare sistemi di calcolo parallelo per queste strutture.

La REDAZIONE

ARCHITETTURE E ALGORITMI SISTOLICI NEL CONTESTO DEL CALCOLO PARALLELO

Stefania Bandini, Massimiliano Goldwurm, Giancarlo Mauri

Dipartimento di Scienze dell'Informazione

Università di Milano

RIASSUNTO

I sistemi sistolici rappresentano una particolare architettura per il calcolo parallelo, costituita da più processori identici collegati secondo strutture geometriche regolari. Questo articolo intende presentare le principali problematiche relative ai sistemi per il calcolo parallelo, collocando i sistemi sistoli in questo quadro generale.

ABSTRACT

Systolic systems are a particular architecture for parallel computing, consisting of a number of identical processor connected in regular geometrical patterns. The paper surveys the main issues about parallel computing, embedding systolic systems into this general frame.

1. INTRODUZIONE

Nei suoi 40 anni di vita, l'informatica ha visto un progresso spettacolare sia nella tecnologia che nella tecnologia di utilizzo degli elaboratori: da un lato con la produzione di componenti e processori sempre più piccoli, veloci e affidabili, e contemporaneamente sempre meno costosi, dall'altro con il costante miglioramento delle tecniche di progetto degli algoritmi e di programmazione. Ciò ha consentito di allargare continuamente il campo di applicazione degli elaboratori, e di affrontare con essi problemi sempre più complessi.

Lo schema concettuale di macchina predominante in questi 40 anni è stato lo schema sequenziale, o "macchina di von Neumann". I punti che caratterizzano la architettura sequenziale sono essenzialmente due:

1) c'è un unico processore che prende le istruzioni una dopo l'altra e le esegue;

2) il processore è fisicamente separato dalla memoria ed è collegato ad essa da lunghe linee di comunicazione (bus).

La scelta di una architettura di questo tipo aveva inizialmente precise motivazioni tecnologiche, in particolare il fatto che le tecnologie di costruzione dei processori e delle memorie erano completamente diverse, e che era necessa-

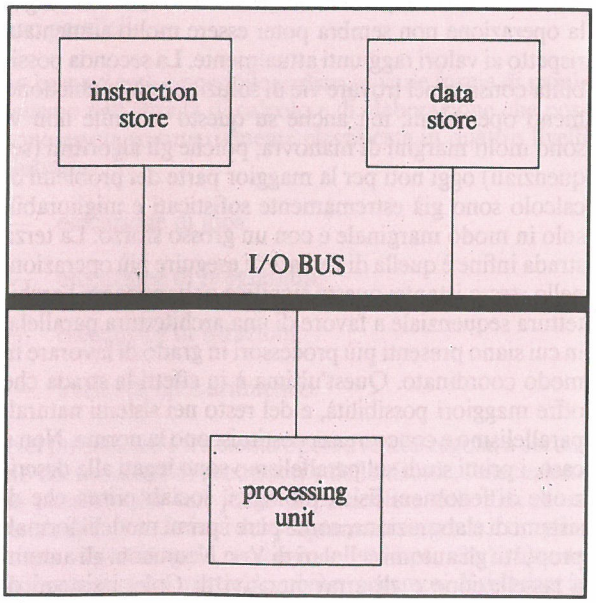

Fig. 1

rio condividere risorse costose come buffers e così via. Inoltre, dal punto di vista della programmazione, la separazione tra memoria e unità centrale semplificava la struttura logica dei programmi e la sostanziale "trasparenza tecnologica" della macchina, cioè la unicità della architettura astratta, indipendentemente dalle tecnologie di realizzazione, facilitava la trasportabilità dei programmi; tale trasportabilità è stata una delle condizioni principali che hanno permesso lo sviluppo e la diffusione del software. D'altro lato, tra gli svantaggi possiamo ricordare la dipendenza delle prestazioni dalla velocità di accesso alla memoria (bottleneck di Von Neumann), l'alto costo di comunicazione dovuto alla lunghezza del bus (ad esempio, se si ha una memoria di M bit organizzati a matrice tridimensionale, la lunghezza media della linea di accesso ad una cella di memoria è proporzionale a $M^{1/3}$), ed infine lo scarso utilizzo "attivo" della memoria: basti pensare che in una memoria di 10^6 bytes un solo byte è attivo in ogni istante.

Il modello della macchina di Von Neumann è rimasto lo schema concettuale di gran parte dei mezzi di calcolo anche quando tecnologie innovative hanno rivoluzionato molte delle singole componenti, ma è oggi chiaro che la potenza di calcolo sviluppabile con architetture di questo tipo ha ormai raggiunto limiti difficilmente superabili, e che alcuni problemi non potranno mai essere affrontati con successo da elaboratori basati su una architettura sequenziale: è il caso, ad esempio, della soluzione di equazioni differenziati per flussi aerodinamici, che richiederebbe velocità di calcolo dell'ordine dei 10 Gflop/s, o del riconoscimento di immagini e di voce.

Ora, se noi dobbiamo risolvere un problema che richiede l'esecuzione di un certo numero di operazioni in un tempo prefissato, e non ci riusciamo, abbiamo tre possibili via d'uscita. La prima consiste nel cercare di impiegare meno tempo per eseguire ogni operazione. Purtroppo, a causa di effettivi limiti fisici, la velocità di esecuzione di una singola operazione non sembra poter essere molto aumentata rispetto ai valori raggiunti attualmente. La seconda possibilità consiste nel trovare vie di soluzione che richiedono meno operazioni; ma anche su questo versante non vi sono molti margini di manovra, poiché gli algoritmi (sequenziali) oggi noti per la maggior parte dei problemi di calcolo sono già estremamente sofisticati e migliorabili solo in modo marginale e con un grosso sforzo. La terza strada infine è quella di cercare di eseguire più operazioni nello stesso istante: questo significa abbandonare l'architettura sequenziale a favore di una architettura parallela, in cui siano presenti più processori in grado di lavorare in modo coordinato. Quest'ultima è in effetti la strada che offre maggiori possibilità, e del resto nei sistemi naturali parallelismo e concorrenza costituiscono la norma. Non a caso, i primi studi sul parallelismo sono legati alla descrizione di fenomeni fisici, biologici, sociali prima che di sistemi di elaborazione, come pure i primi modelli formali proposti: gli automi cellulari di Von Neumann, gli automi a tassellazione e gli array iterativi di Cole, i sistemi di Lindenmayer, le reti di Petri.

2. TECNOLOGIA E CALCOLO PARALLELO

Già Babbage, 150 anni fa, nel suo "Analogical Engine" (1842), aveva intuito le possibilità offerte dal parallelismo per il calcolo, e algoritmi paralleli sono stati proposti ben prima che esistessero macchine reali in grado di computarli.

Nei sistemi di calcolo, il parallelismo ha fatto la sua comparsa in forme limitate, ma via via sempre più estese, fin dagli anni '50: dalla sovrapposizione delle operazioni di I/O alla esecuzione di più programmi, dall'uso di più unità aritmetiche nell'esecuzione di operazioni, alla costruzione di reti di processori variamente interconnessi. Del resto, se si confrontano gli elaboratori del 1950 con quelli del 1975, si scopre, a fronte di un aumento della velocità di propagazione dei segnali di circa un fattore 10^3, un incremento della velocità complessiva di calcolo di un fattore compreso tra 10^5 e 10^6, passando per esempio dalle 100 operazioni aritmetiche al secondo su numeri a 32 bit in virgola fissa, realizzate su EDSAC, ai 130 milioni di operazioni aritmetiche al secondo su numeri a 64 bit in virgola mobile, realizzate su CRAY1. È evidente che la differenza tra l'incremento della velocità di calcolo e quello della velocità di propagazione dei segnali elettrici è dovuta in buona parte al sempre maggiore grado di parallelismo introdotto nell'attività di calcolo.

In generale, i calcolatori della prima generazione, quelli basati su circuiti a valvole, riproducevano fedelmente l'architettura di Von Neumann, realizzando ciascuna operazione (memorizzazione, lettura dei dati in memoria, operazioni in input e di output, operazioni logiche e aritmetiche) in modo strettamente sequenziale. Con la realizzazione e la commercializzazione dei transistor (fine anni '50) si affermano sul piano produttivo e commerciale macchine che permettono anche varie forme di parallelismo sia nella trasmissione dei dati che nell'esecuzione di programmi. Ma è solo con l'introduzione delle tecnologie VLSI (Very Large scale Integration), verso la fine degli anni '70, che diviene tecnologicamente possibile la realizzazione di sistemi che permettono un massiccio uso di forme di parallelismo esplicito, gestibile direttamente dal programmatore. Uno dei principali vantaggi offerti dalla tecnologia VLSI [8] è la disponibilità di grandi quantità di hardware a basso costo, e soprattutto senza distinzione tecnologica tra memoria e processori: basti pensare che dal 1970 ad oggi si è assistito, come si può vedere dalla tabella seguente, ad una crescita esponenziale del numero di elementi attivi in un chip di area dell'ordine del cm^2, con una loro decuplicazione ogni cinque anni.

1970	10^3
1975	10^4
1980	10^5
1985	10^6

Inoltre, i circuiti VLSI offrono dimensioni e dissipazione

di potenza ridotte e maggiore affidabilità circuitale.

3. PARADIGMI DI COMPUTAZIONE PARALLELA

Se da un alto il parallelismo permette di ottenere sistemi con maggior throughput, maggiore affidabilità, maggiore efficienza e velocità di calcolo, dall'altro esso introduce, a vari livelli, elementi di complessità decisamente superiori a quelli che si ritrovano nel calcolo sequenziale. In particolare, per sfruttare a fondo le possibilità offerte dal parallelismo occorrono nuove idee su come organizzare in termini architetturali computers paralleli, su quali modelli computazionali astratti assumere, ed infine sulle metodologie di disegno di algoritmi e di programmazione adatte al calcolo parallelo. In questo paragrafo e nei prossimi cercheremo di capire queste problematiche, a partire dalle diverse forme di parallelismo che si possono presentare [3].

Una semplice forma di parallelizzazione è rappresentata dal **pipelining,** cioè dalla segmentazione di un processo in sottoprocessi ognuno dei quali è eseguito da un processore che passa poi i risultati al processore responsabile del sottoprocesso successivo. Se il processo deve assere eseguito più volte su dati diversi, si realizza così una vera e propria "catena di montaggio", o **pipeline,** con notevoli risparmi di tempo. Naturalmente, per uno sfruttamento ottimale della catena è necessario che la suddivisione dei compiti sia equilibrata, cioè che i singoli sottoprocessi richiedano approssimativamente lo stesso tempo. In questo modo non accade che un processore, dopo aver portato a termine il suo compito, debba rimanere inattivo in attesa di ricevere un nuovo input dal processore precedente.

Come esempio, consideriamo l'addizione (elemento per elemento) di due vettori **a** e **b** di numeri in virgola mobile. L'addizione di due elementi a_i, b_i dei vettori può essere decomposta in 5 fasi:

a) normalizzazione di a_i, b_i;

b) confronto degli esponenti delle forme normalizzate;

c) "allineamento" tramite shift della mantissa dell'operando minore;

d) addizione delle mantisse;

e) normalizzazione del risultato.

Potremo allora utilizzare una catena di cinque processori:

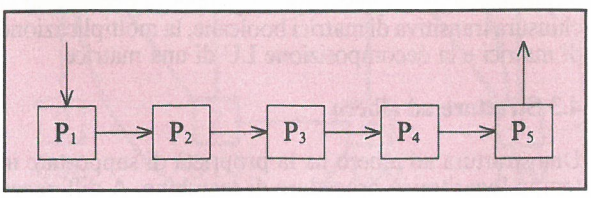

Fig. 2

In questa catena P_1, dopo aver normalizzato a_1 e b_1, li passa a P_2 per il confronto degli esponenti, e contemporaneamente inizia a normalizzare a_2, b_2, e così via. Al quinto passo tutti i processori saranno impegnati (se il vettore ha lunghezza ≥ 5) e si avrà il massimo parallelismo. Il pipelining è efficace soprattutto in applicazioni specializzate che consentono l'uso di un grande numero di processori: è il caso di numerosi problemi relativi alla elaborazione di segnali, come il campionamento digitale, l'analisi in frequenza, il filtraggio.

A questo stesso esempio è applicabile un diverso paradigma, quello di **parallelismo vettoriale:** esso si basa sulla disponibilità di un array di processori identici che eseguono ad ogni istante la stessa operazione du diversi elementi di un array di dati, sotto un controllo comune.

Un terzo paradigma di parallelizzazione è rappresentato dal **parallelismo funzionale.** Esso prevede l'utilizzo di più unità indipendenti per l'esecuzione di diverse funzioni; ad esempio, la gestione dell'input/output con processori separati dalla unità centrale è una forma di parallelismo funzionale presente fin dalle primissime fasi di sviluppo dell'informatica. Tipiche di questa situazione sono anche le varie unità di calcolo "specializzate" che realizzano le diverse operazioni logiche e aritmetiche (addizioni, moltiplicazioni, etc.).

Infine, la forma più completa di parallelismo è rappresentata dal **multiprocessing,** in cui sono presenti più processori ognuno dei quali è in grado di eseguire un proprio programma, diverso da quello degli altri; ogni processore può inoltre comunicare con tutti gli altri attraverso una memoria condivisa oppure una rete di comunicazione opportuna. Il *multiprocessing* ha assunto una importanza particolare da quando lo sviluppo delle tecnologie VLSI ha permesso di ridurre drasticamente il costo di realizzazione dei microprocessori.

In genere, questi diversi paradigmi di parallelizzazione sono presenti, con diversi pesi relativi, in tutti gli elaboratori paralleli.

In base ad essi, è possibile realizzare varie forme di parallelismo nell'attività di calcolo e di elaborazione che possono essere sommariamente classificate in quattro livelli distinti:

— esecuzione di jobs;

— esecuzione di programmi;

— esecuzione di istruzioni;

— livello di bit e aritmetico.

Nel primo caso è il sistema operativo che regola, a secondo del numero dei processori e delle risorse, l'esecuzione in parallelo dei vari job e in particolare delle fasi di un medesimo job (lettura del programma sorgente, compilazione, assemblaggio, esecuzione, stampa, etc.). Questo può avvenire permettendo a più programmi di risiedere in memoria centrale e assegnando alternativamente a questi

le varie risorse del sistema (unità di calcolo, canali I/O, memoria di massa, etc.).

Al livello successivo è possibile prevedere l'esecuzione in parallelo di parti indipendenti di un medesimo programma opportunamente riconosciute in una fase di analisi logica. Per esempio, se i diversi cicli di un loop sono indipendenti tra loro, è possibile assegnare l'esecuzione dei vari cicli a unità di calcolo distinte che lavorano in parallelo, e unificare poi opportunamente i risultati. In taluni casi, inoltre, la possibilità di utilizzare istruzioni macchina a vettore, in grado, cioè, di modificare nello stesso modo e contemporaneamente un vettore di numeri, ha consentito di eseguire mediante un numero ridotto di istruzioni quei loop che possono essere visti come modifica di un vettore a numeri.

A livello di istruzione è possibile considerare le sotto-operazioni corrispondenti e prevedere forme di pipelining che consentano di sovrapporre l'esecuzione di queste ultime. Questo particolare uso del metodo di pipelining per eseguire una istruzione era tipico soprattutto dei processori scalari veloci (fast scalar computers) che non disponevano ancora di istruzioni a vettore e operavano direttamente solo sulla singola parola.

Infine, al livello più basso, è possibile operare in parallelo su tutti o su una parte dei bit di una parola; inoltre l'esistenza di istruzioni macchina a vettore consente di fatto la modifica in parallelo dei bit di più parole. Già i primi calcolatori prevedevano la possibilità di operare in parallelo sui bit di una parola; tuttavia, è con lo sviluppo più recente della tecnologia VLSI [8] e la realizzazione dei primi microprocessori su chip l'attività in parallelo a tale livello acquista una importanza particolare.

4. ASPETTI ARCHITETTURALI

La presenza di diversi processori in un elaboratore parallelo comporta scelte critiche sul numero e sul tipo di processori, ma soprattutto su come sono collegati tra di loro, ai moduli di memoria ed ai canali di I/O.

Potremo descrivere questi collegamenti con un grafo orientato $G = \langle V,E \rangle$, in cui i vertici rappresentano i moduli o processori e gli archi i loro collegamenti.

La maggior potenza e flessibilità si può ottenere collegando tra loro n CPU complete, ognuna con una sua memoria, in modo che ognuna possa comunicare direttamente con ogni altra. Questa forma di collegamento totale comporterebbe n^2 connessioni, mentre ogni nodo del grafo avrebbe fan-in n-1. Poiché un tale schema non è tecnologicamente realizzabile a causa della crescita troppo rapida del numero di connessioni e del fan-in dei nodi, si presenta la necessità di trovare architetture in cui fan-in dei nodi sia costante o al più cresca logaritmicamente rispetto al numero dei nodi. Generalmente architetture di questo tipo sono adatte alla implementazione di particolari paradigmi algoritmici; naturalmente, siamo interessati ad architetture che coprano una classe sufficientemente ampia di algoritmi, e siano quindi intermedie tra le architetture universali o general-purpose e le architetture special-purpose: si parla in questi casi di architettute **broad-purpose.** Vediamo ora alcune delle più diffuse geometrie di comunicazione.

4.1 Array lineari unidimensionli

L'array lineare unidimensionale, insieme di processori ordinati in sequenza ognuno dei quali è collegato al precedente e al successivo, rappresenta la più semplice e fondamentale geometria di processori connessi. I dati possono fluire in una sola direzione o in ambedue le direzioni simultaneamente, a seconda dell'algoritmo. Questa architettura è particolarmente adatta per operazioni di filtraggio, la convoluzione, la trasformata di Fourier veloce, la moltiplicazione matrice-vettore, l'ordinamento on-line, la priority queue, le unità aritmetiche pipelinizzate.

4.2 Array bidimensionali

In una geometria di interconnessione bidimensionale ogni processore comunica con un numero di processori adiacenti che dipende dal tipo di geometria definita: quattro per il quadrato, sei per l'esagono, etc..

L'array quadrato è una delle geometrie di comunicazione più studiate nell'ambito della processazione parallela, e molta dell'attività di ricerca riguardante lo studio di algoritmi paralleli per geometrie di comunicazione di questo tipo sono state influenzate dalla realizzazione dell'ILLIAC IV, che possiede un array di processori 8*8 (vedi paragrafo 7).

Un esempio di algoritmo parallelo per questo tipo di geometria di comunicazione è rappresentato dall'algoritmo Bitonic Sort. Altri algoritmi adatti a questa architettura sono la chiusura transitiva, la moltiplicazione di matrici, alcuni algoritmi su grafi.

Un'altra struttura bidimensionale spesso usata è l'array esagonale. Il processore di base che compone strutture bidimensionali esagonali ha 3 registri, R_A, R_B, R_C e 6 connessioni esterne, 3 per la comunicazione in ingresso e 3 per la comunicazione in uscita. In ogni unità di tempo il processore invia i dati dalle sue linee di ingresso, denotate da A, B, C rispettivamente nei registri R_A, R_B, R_C e computa $R_C \leftarrow\!\!- R_C + R_A * R_B$, rendendo disponibili i valori di input in R_A e R_B insieme al nuovo valore di R_C come uscita sulle linee di output ancora denotate, rispettivamente, da A, B, C. Per quanto riguarda gli algoritmi implementabili su questo tipo di geometria ricordiamo la chiusura transitiva di matrici booleane, la moltiplicazione di matrici e la decomposizione LU di una matrice.

4.3 Strutture ad albero

Una struttura ad albero ha la proprietà di supportare in tempo logaritmico procedure di searching. A differenza delle strutture ad array le connessioni della struttura ad albero non sono uniformi; infatti, la distanza tra due processori connessi decresce muovendosi verso la radice.

Nell'implementazione su chip di tali strutture il tempo che impiega un segnale per propagarsi lungo un filo non può essere indipendente dalla lunghezza del filo. Tuttavia, 'uso di driver appropriati permette di mantenere costanti i tempi di comunicazione tra un processore e l'altro. In questo modo si conserva la proprietà logaritmica dei tempi di comunicazione nella struttura ad albero. Si può addirittura assumere che il tempo per inviare un insieme di dati attraverso ogni link nell'albero sia costante e che la radice sia un nodo I/O per le comunicazioni con il mondo esterno. Inoltre, anche usando driver di maggiori dimensioni è possibile realizzare layout di tali strutture che occupano un'area proporzionale al numero dei nodi dell'albero.

Un difetto può essere rappresentato dal fatto che i processori situati ad un alto livello dell'albero possono diventare colli di bottiglia se la maggioranza delle comunicazioni non sono adiacenti ai processori di basso livello. Gli algoritmi paralleli che utilizzano strutture ad albero devono essere in grado di ovviare a questo inconveniente inerente alla geometria di comunicazione stessa. Tra questi vi sono algoritmi di ricerca e di valutazione parallela di funzioni.

4.4 Architetture a shuffle

Un'architettura particolarmente importante è descritta mediante i grafi di **perfect-shuffle.** Si tratta di una rete di n processori $P_0, P_1,..., P_{n-1}$ tale che il processore di indirizzo binario $i_m i_{m-1}...i_1$ (m=log n), ha connessioni con i nodi di indirizzo:

$S(i_m i_{m-1}...i_1) = i_{m-1} i_{m-2}...i_1 i_m$

$U(i_m...i_1) = i_1 i_m...i_2$

$B(i_m i_{m-1}...i_1) = i_m i_{m-1}...\bar{i}_1$ dove $\bar{i}_1$ è il complemento di i_1.

Nel caso di 8 processori si ha:

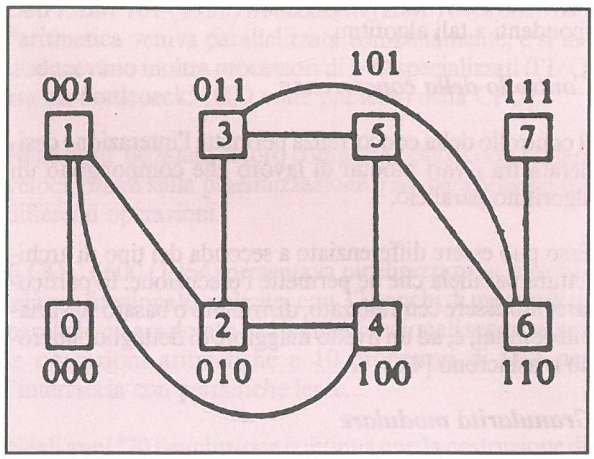

Fig. 3

Questa architettura è poco regolare, poiché presenta connessioni di lunghezza molto variabile, ma ha una grande capacità di effettuare permutazioni, ed è quindi particolarmente adatta per il sorting (Bitonic-Sort). Inoltre, l'importanza di tale struttura risiede nel fatto che essa può venire impiegata per simulare strutture più complesse (ad es. ipercubo) che non sono però direttamente realizzabili in termini di lay-out per l'elevato fan-in dei nodi.

4.5 Strutture n-dimensionali: ipercubo

Un'altra struttura di comunicazione rilevante è rappresentata da architetture n-dimensionali, come l'ipercubo. tale geometria di comunicazione è definita da un grafo G= <V,E> dove V è l'insieme dei vertici $(a_1,...,a_n)$, ogni a_i appartiene all'insieme (0,1), mentre E è l'insieme dei lati $((a_1,...,a_n),(b_1,...,b_n))$ per i quali esiste un solo j tale che $b_j \neq a_j$. Molti algoritmi sono implementabili su questa struttura, ad esempio, la trasformata di Fourier, l'algoritmo di convoluzione, alcuni algoritmi di sorting. A causa dell'elevato fan-in dei nodi, inoltre, esso rappresenta un modello di comunicazione per molti algoritmi paralleli, ma non si presta ad una diretta implementazione; per questo motivo si tende a determinare più semplici architetture con fan-in più limitati che simulano l'ipercubo. Tra questi, ricordiamo l'ipercubo a cicli (CCC) o le strutture di tipo shuffle.
L'architettura CCC [9] è una architettura a ipercubo a n vertici in cui ogni vertice è costituito da un gruppo di log n moduli connessi ciclicamente. Ogni modulo ha tre porte: P (Predecessore), che lo connette col modulo precedente nel ciclo, S (Successore) che lo connette col modulo seguente, ed E (Esterno), che lo connette con un modulo appartenente ad un vertice adiacente dell'ipercubo. Ad esempio, il CCC di otto vertici ha la struttura seguente:

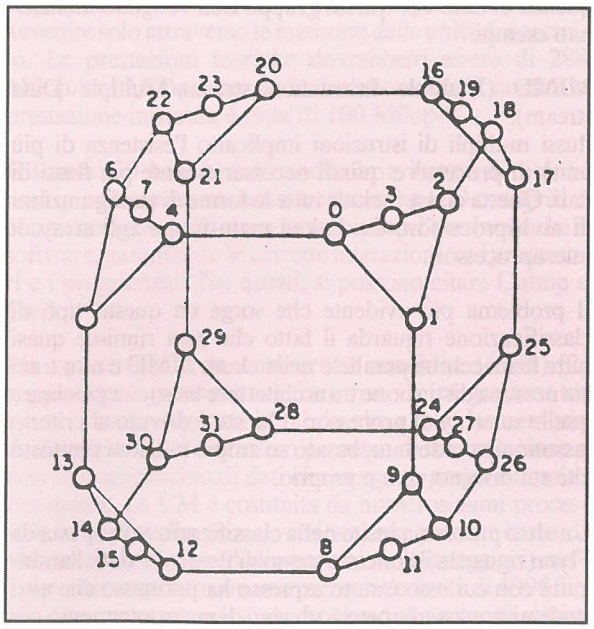

Fig. 4

5. LA TASSONOMIA DI FLYNN

Il criterio tassonomico adottato da Flynn [1] per esplicitare una classificazione delle architetture parallele è basato sulla relazione che in una macchina intercorre tra le istruzioni e i dati che devono essere processati. In questo contesto è basilare il concetto di flusso (stream), definito come una sequenza di item (istruzioni o dati) eseguita o operata da un processore. Il criterio di raggruppamento in classi a partire da questo concetto di basa sul fatto che i flussi di dati o di istruzioni siano singoli o multipli; emergono in questo modo quattro classi distinte:

SISD (Single Instruction stream/Single Data stream):
è la macchina seriale di Von Neumann, in cui c'è solo un flusso di istruzioni (e quindi un'unica unità di processo d'istruzioni) e ognuna delle istruzioni aritmetiche istanzia una operazione aritmetica che guida un singolo flusso di dati.

SIMD (Single Instruction stream/Multiple Data stream):
è una macchina che conserva un singolo flusso di istruzioni, ma ha istruzioni vettoriali che danno origine a operazioni plurime. Ogni elemento del vettore è visto come un membro di un flusso separato di dati e dunque ci sono flussi di dati multipli. Proprio per i criteri della tassonomia di Flynn, per questa classe di macchine risulta irrilevante che la capacità di processo del vettore sia realizzata mediante pipelining o ricorrendo ad array di processori. Esempi di macchine che cadono in questo gruppo sono CRAY1 (vettoriale pipelined) o ILLIAC IV (array di processori).

MISD (Multiple Instruction stream/Single Data stream):
questa classe comprende le architetture in cui più istruzioni operano simultaneamente su di un unico dato. Essa include anche organizzazioni specializzate di flusso in cui flussi multipli di istruzioni agiscono su una singola sequenza di dati. Per questo gruppo non vengono menzionati esempi.

MIMD (Multiple Instruction stream/Multiple Data stream):
flussi multipli di istruzioni implicano l'esistenza di più unità di processo e quindi necessariamente più flussi di dati. Questa classe include tutte le forme di configurazioni di multiprocessori, dai linked main-frame agli array di microprocessori.

Il problema più evidente che sorge da questo tipo di classificazione riguarda il fatto che essa riunisce quasi tutte le macchine parallele nella classe SIMD e non traccia nessuna distinzione tra architetture basate su pipeline e quelle su array di processori. Questo è dovuto al criterio tassonomico adottato, basato su ampie funzioni piuttosto che sul disegno vero e proprio.

Un altro problema insito nella classificazione proposta da Flynn riguarda il concetto stesso di flusso dei dati: l'ambiguità con cui esso è stato espresso ha permesso che altri studiosi trovassero metodi plurimi di raggruppamento per le macchine a vettore pipelined, non unicamente nella classe SIMD, dove Flynn le aveva esplicitamente collocate. Ad esempio, considerando un vettore come dato singolo una macchina SIMD diventa SISD.

6. ASPETTI ALGORITMICI

La possibilità di usare in modo ottimale gli elaboratori paralleli, sfruttandone fino in fondo le capacità, dipende dalla disponibilità di tecniche adeguate per il progetto di algoritmi e il controllo della concorrenza. L'obiettivo fondamentale nel disegno di algoritmi paralleli è quello di mantenere attivo ad ogni istante il maggior numero possibile di processori e localizzarli il più possibile vicino alle informazioni che essi richiedono. Inoltre, è necessario ripartire equamente il carico di lavoro tra i diversi processori. È allora evidente che le scelte architetturali influenzano fortemente il disegno degli algoritmi.

Poiché gli algoritmi non sono indipendenti dalla architettura, si perdono alcuni dei vantaggi della architettura di Von Neumann, legati alla "trasparenza" tecnologica della macchina (il programmatore non "vede" la macchina o, meglio, tutti i programmatori "vedono" la stessa macchina), in particolare la trasportabilità del software.

Un altro problema riguarda la possibilità di parallelizzare software sequenziale; questo non sempre è possibile, e vi sono alcuni esempi, legati per esempio alle tecniche greedy, che mostrano come la parallelizzazione di algoritmi sequenziali anche ottimali possa portare ad algoritmi paralleli molto peggiori di algoritmi "pensati" direttamente in parallelo.

6.1 Tassonomia degli algoritmi paralleli

Si può definire un algoritmo parallelo come un insieme di moduli di lavoro indipendenti che possono essere eseguiti in parallelo e che comunicano tra loro durante la fase di esecuzione. Diremo che una architettura parallela G=<V> supporta un algoritmo A se E contiene un arco per ogni trasferimento di dati nella esecuzione di A.

È possibile individuare tre dimensioni dello spazio degli algoritmi paralleli e caratterizzare le architetture corrispondenti a tali algoritmi.

Controllo della concorrenza

Il controllo della concorrenza permette l'interazione desiderata tra i vari moduli di lavoro che compongono un algoritmo parallelo.

Esso può essere differenziato a seconda del tipo di architettura parallela che ne permette l'esecuzione: in particolare può essere centralizzato, distribuito o basato su variabili comuni, e, ad un livello maggiore di dettaglio, sincrono o asincrono [4].

Granularità modulare

La granularità di un algoritmo parallelo viene definita-

intuitivamente come la quantità massima di computazione che un modulo di lavoro "ideale" può fare prima di comunicare con altri moduli. In questo modo la granularità di un algoritmo parallelo individua il livello di intensità di comunicazione dell'algoritmo stesso: ad esempio, una bassa granularità modulare esige una comunicazione intramodulare molto frequente.

Geometria di comunicazione

Le connessioni tra i moduli di lavoro in cui è decomposto un algoritmo parallelo sono organizzate secondo una precisa geometria, proprio come i processori.

Possiamo a questo punto individuare lo spazio S_a degli algoritmi paralleli come un prodotto cartesiano tra l'insieme (C_o) di controlli della concorrenza, l'insieme (Mg) delle granularità e l'insieme (Cg) delle geometrie di comunicazione:

$$S_a = (C_o) \text{ X } (M_g) \text{ X } (C_g)$$

Per quanto riguarda la associazione tra algoritmi ed architetture parallele, è possibile vedere che macchine SIMD corrispondono ad algoritmi che richiedono un controllo centralizzato, mentre a macchine MIMD corrispondono algoritmi asincroni con una granularità relativamente grande. Infine, nel valutare la complessità di un algoritmo parallelo, è necessario tener conto, oltre al tempo di esecuzione per un dato di dimensione prefissata n, anche il numero di processori richiesto, l'area occupata da questi nel caso di circuiti VLSI e la lunghezza dei fili.

7. LE REALIZZAZIONI

I primi calcolatori avevano aritmetica totalmente seriale. Già nel 1951 tuttavia, con ACE, l'aritmetica veniva parzialmente parallelizzata, consentendo operazioni in parallelo sui 32 bit della parola, e comparivano moduli (lettore-perforatore schede; moltiplicatore; divisore) in grado di operare in parallelo con altre parti della macchina (parallelismo funzionale) [3].

Con l'IBM 701 (1953) e successori (IBM 704, IBM 709) l'aritmetica veniva parallelizzata completamente, e si introducevano inoltre processori di I/O specializzati (l'I/O era un bottleneck: 1000 volte più lento della CPU).

Negli anni '60 fanno la loro comparsa i calcolatori scalari veloci, basati sulla pipelinizzazione e sul parallelismo tra differenti operazioni.

Il CDC 6600 (1964) presentava pipelinizzazione e parallelismo funzionale esplicito, con 32 banchi di memoria in parallelo ed era dotato di 10 unità funzionali separate per le operazioni aritmetiche e 10 processori di I/O per l'interfaccia con periferiche lente.

Negli anni '70 l'evoluzione continua con la costruzione di elaboratori vettoriali pipelinizzati e macchine ad array.

La macchina più significativa è ILLIAC IV (1971), progettato per la soluzione di equazioni differenziali alle derivate parziali, in particolare per problemi metereologici. Il progetto prevedeva 256 processori floating point, organizzati in quattro blocchi di 64 (array 8x8), ognuno con memoria privata da 2000 parole a 64 bit e unità di controllo propria. I blocchi avrebbero dovuto essere connessi da bus di I/O altamente paralleli, con un processore centrale come supervisore. Questo schema architetturale era poco praticabile negli anni '70, essendo troppo avanzato rispetto alla tecnologia disponibile. Proprio per questo è stato possibile realizzare un solo blocco, e raggiungere velocità di 50 Mflop/s contro una velocità prevista per la macchina completa di 1 Gflop/s. Tuttavia l'attività intorno ad ILLIAC è stata di notevole importanza anche per lo sviluppo di linguaggi per il calcolo parallelo, ed ha portato ai primi tentativi di definizione di primitive per il parallelismo.

Un secondo elaboratore di grande successo è CRAY 1 (1976), costituito da 12 unità funzionali pipelinizzate, con ciclo di 12.5 ns, 16 banchi di memoria da 1 Mword, 8 registri vettoriali da 64 numeri fp di 64 bit, 32 istruzioni macchina per operare su tali vettori, 3 unità funzionali riservate per operazioni vettoriale ed altre tre condivise con le operazioni scalari. Le prestazioni di questa macchina raggiungono i 130 Mflop/s.

Il CDC STAR 100 (1973) permetteva il trattamento di vettori di grandi dimensioni con primitive APL, con velocità dell'ordine di 100 Mflp/s.

PEPE, sviluppato per gestire rivelatori radar di missili, è costituito da 299 unità di elaborazione, ognuna di tre processori, per l'input dei segnali radar, il data processing e l'output di segnali di controllo, rispettivamente, con una unità di controllo centrale per ogni tipo di processore ed un CDC 7600 come host. Ogni unità di elaborazione è responsabile di un obiettivo, ed i collegamenti possono avvenire solo attraverso le memorie delle unità di controllo. Le prestazioni teoriche dovrebbero essere di 288 Mflop/s (1 Mflop/s per unità di elaborazione), mentre la prestazione misurata è stata di 100 Mflop/s.

Gli anni '70 hanno visto anche lo sviluppo di sistemi multiprocessore con memoria e accesso a I/O condivisi, e con sistemi operativi che controllano sia l'hardware che il software, garantendo le corrette interazioni tra i processori e i programmi. Tra questi, si possono citare Cmmp e Cm*.

Più recentemente, la **connection machine** (1985) [2] rappresenta il passo più avanzato verso la integrazione tra memoria e processore, in modo che anche la memoria risulti parte attiva dell'intero processo di computazione e non solo magazzino di dati che vengono prelevati molto raramente. La CM è costituita da numerosissimi processori (65.536 nella prima versione, ma sono in corso di realizzazione versioni di dimensioni molto maggiori) molto semplici, ognuno con una memoria privata di 4096 bit. L'I/O avviene attraverso un canale ad alta velocità gestito da uno host, che provvede anche al controllo; vi è

un'unica sequenza di istruzioni, ma l'aspetto più interessante sta nel fatto che le connessioni sono programmabili, in modo che la topologia della macchina possa adeguarsi alla topologia del problema.

8. IL PARADIGMA SISTOLICO

Un posto particolare tra le architetture parallele hanno i sistemi sistolici, che verranno meglio descritti nell'articolo successivo. Citando Kung [5], "systolic systems are an attempt to capture the concepts of parallelism, pipelining and interconnection structures in a unified framework of mathematics and engineering". Essi rappresentano una classe di "macchine su un solo chip" costituite da una sola rete di processori semplici e primitivi, con una struttura regolare in cui i dati fluiscono con ritmo regolare, in cui si fondono i paradigmi di pipelining e multiprocessing. Il controllo della concorrenza è centralizzato e sincrono, con un clock comune.

L'aggettivo "sistolico" è stato preso in prestito dalla fisiologia in cui questa parola viene impiegata per denotare le contrazioni che ricorrono ritmicamente nel cuore e che permettono al sangue di fluire alla periferia per via arteriosa. In una macchina sistolica la funzione di ogni processore è analoga a quella del cuore: ogni processore regolarmente "pompa" dati dentro e fuori ed ogni volta esegue qualche computazione. Il flusso costante di dati è analogo alla pressione sanguigna.

Il controllo della computazione e della comunicazione è molto semplice, essendo sostanzialmente predeterminato dalla geometria della rete di processori; la comunicazione è locale, tra processori contigui, ed avviene in tempo unitario; la computazione avviene all'interno di ogni processore ad ogni segnale di clock, in modo ritmico e perpetuo; gli operandi arrivano insieme al processore, per cui non vi sono attese o tempi morti, a patto, naturalmente, che il sistema sistolico sia progettato in modo tale da garantire che i dati giusti arrivino al processore giusto nel momento giusto: questa è la difficoltà principale nel progetto di sistemi sistolici.

Più precisamente, gli elementi chiave da tener presenti nel progetto di un sistolico sono tre:

— determinazione di una struttura geometrica semplice e regolare. Questo consente di ridurre i costi di disegno, attraverso l'uso di un numero ridotto di sottostrutture semplici usate in modo ripetitivo con interfacce semplici, e di ottenere implementazioni in chip di basso costo e di alte prestazioni;

— disegno di una rete di comunicazione e controllo regolare, in modo da ottenere un alto grado di concorrenza con una implementazione efficiente;

— bilanciamento tra computazione e I/O, in modo da ottenere il massimo speed-up compatibile con la larghezza di banda dell'I/O disponibile.

Dal punto di vista delle associazioni tra macchina parallele ed algoritmi, la macchine sistoliche corrispondono ad algoritmi sincroni che impiegano un controllo distribuito ottenuto da un semplice meccanismo di controllo locale ed hanno una granularità modulare bassa e costante. Esse rappresentano la architettura in cui è maggiormente evidente la interdipendenza e la sinergia tra struttura della macchina e struttura dell'algoritmo. Proprio per questo, i sistemi sistolici sono stati concepiti essenzialmente come sistemi special-purpose, da inserire, al limite sotto forma di scheda, in uno host per l'esecuzione di calcoli particolari (ad esempio prodotti di matrici) ad altissima velocità. Tuttavia recentemente sono stati proposti alcuni sistolici programmabili; tra essi ricordiamo il PSC, sviluppato presso la Carnegie Mellon University, sistolico programmabile con un controllo distribuito e comunicazione in rete, e il GAPP (Geometric Arithmetic Parallel Processor), di cui si parlerà ampiamente in due articoli successivi.

I campi applicativi cui meglio si adatta l'uso di array sistolici sono l'elaborazione di immagini e la computer vision, la fisica nucleare, l'analisi di struttura, il riconoscimento del parlato, l'elaborazione di dati da radar o sonar o di segnali medici, le meteorologia.

9. BIBLIOGRAFIA

[1] M. Flynn, SOME COMPUTER ORGANIZATIONS AND THEIR EFFECTIVENESS, IEEE Trans. on Computers, C-21, 1972

[2] D. Hillis, THE CONNECTION MACHINE, MIT Press, 1985

[3] R. Hockney, C. Jessope, PARALLEL COMPUTERS, Adam Hilger Ltd., Bristol, 1981

[4] H.T. Kung, THE STRUCTURE OF PARALLEL ALGORITHMS, Advances in Computer, vol. 19, 1981

[5] H.T. Kung, WHY SYSTOLIC ARCHITECTURES? COMPUTER, vol. 15, no. 1, 1982

[6] L. Kronsjö, COMPUTATIONAL COMPLEXITY OF SEQUENTIAL AND PARALLEL ALGORITHMS, J. Wiley and Son, 1986

[7] C.E. Leiserson, AREA EFFICIENT VLSI COMPUTATIONS, MIT Press, 1983

[8] C.A. Mead, L.A. Conway, INTRODUZIONE AI SISTEMI VLSI, F. Angeli, 1984

[9] F.P. Preparata. J.E. Vuillemin, THE CUBE CONNECTED CYCLES: A VERSATILE NETWORK FOR PARALLEL COMPUTATION, Proc. 20th Annual IEEE Symp. on Foundations of Computer Science, 1979

[10] J.D. Ullman, COMPUTATIONAL ASPECTS OF VLSI, Computer Science Press, 1984

ARCHITETTURE SISTOLICHE

Roberto Grande

Dipartimento di Scienze dell'Informazione

Università di Milano

RIASSUNTO

In questo articolo si vogliono mostrare attraverso degli esempi le caratteristiche peculiari e le potenzialità delle architetture sistoliche. Nella prima parte sono descritte in modo schematico ed informale le proprietà ed i vincoli che caratterizzano le architetture sistoliche, mettendo in evidenza come questi soddisfino alle richieste implementative e progettuali avanzate della tecnologia VLSI. Nella seconda parte vengono presentati degli esempi paradigmatici dell'approccio sistolico: convoluzione, correlazione, chiusura transitiva; infine viene mostrata la trasformazione di algoritmo semisistolico in un algoritmo sistolico puro.

ABSTRACT

The aim of this work is to show through same examples the power of systolic architectures and their most interesting features.
The first part of the paper presents clearly and informally features and requirements of systolic architectures.
The presentation is oriented to show how these requirements match those of VLSI technology.
The second part is devoted to some classical examples: convolution, correlation, transitive closure.
Finally it is explained how to transform a "semisystolic" algorithm into an equivalent, purely systolic one.

1. INTRODUZIONE

In questo articolo si vogliono mostrare attraverso degli esempi le caratteristiche peculiari e le potenzialità delle architetture sistoliche.

Prima di procedere oltre sono necessarie delle osservazioni preliminari.

Usualmente si possono distinguere tre livelli di astrazione a cui fare riferimento nella definizione di un sistema computazionale: a un primo livello si ha il "modello computazionale" in cui sono specificati concettualmente gli elementi della computazione, le loro relazioni, i vincoli a cui devono sottostare e le operazioni elementari che possono essere compiute. Ad un secondo livello c'è l'"architettura astratta" che implementa un modello computazionale; diverse architetture astratte possono essere adottate per l'implementazione di uno stesso modello computazionale. Infine si colloca il livello dell'"algoritmo" o "computazionale" che è un programma per l'architettura astratta; una stessa architettura astratta supporta più algoritmi.

Nel caso dei Sistemi Sistolici l'interazione tra questi livelli è strettissima, al punto da rendere spesso artificiosa la distinzione tra gli ultimi due. La maggior parte dei parametri di un algoritmo, infatti, sono qui già fissati al livello dell'architttura astratta.

I Sistemi Sistolici nascono come implementazione diretta di particolari algoritmi e, quindi, essenzialmente come sistemi "special purpose" o "algorithm oriented".

Recentemente lo sviluppo e la commercializzazione di processori sistolici programmabili ha portato a sistemi più flessibili con un più ampio range di applicazioni; tuttavia si hanno generalmente prestazioni efficienti solo per classi relativamente ristrette di algoritmi. Quanto detto giustifica la consuetudine, che spesso si riscontra in letteratura, di descrivere contemporaneamente l'algoritmo sistolico e l'architettura astratta che 'più direttamente' lo implementa. In questo articolo, a meno che non sia specificato altrimenti, ci riferiremo ad algoritmi sistolici e alla loro

'più diretta' implementazione in una architettura astratta di processori sistolici ideali, cioé una architettura che mantenga una corrispondenza diretta tra le operazioni e le variabili dell'algoritmo e, rispettivamente, i processori e le linee di connessione dell'architettura sistolica.

Nei precedenti interventi si è visto quali sono le proprietà formali che caratterizzano un sistema sistolico, quali sono i vincoli a cui deve soddisfare; in breve si è delineato il "modello computazionale" proprio dei sistemi sistolici.

Questo intervento è strutturato nel seguente modo: nella prima parte si ha una rassegna informale delle priorità e dei vincoli tipici di tali sistemi allo scopo di mettere in evidenza come questi soddisfino alle richieste implementative e progettuali avanzate dalla tecnologia VLSI. In una seconda parte verranno presentati degli esempi che si possono considerare in un certo senso 'paradigmatici' dell'approccio sistolico e che hanno lo scopo di illustrare e precisare le argomentazioni precedentemente fatte.

2. I SISTEMI SISTOLICI

Per facilitare la collocazione dei sistemi sistolici nello spazio delle architetture parallele conviene adottare alcuni criteri generali di classificazione:

- Qualità dei processori:
  - non omogenei;
  - omogenei:
    - unità funzionali specializzate;
    - processori general purpose.

- Controllo della concorrenza:
  - centralizzato: sincrono;
  - distribuito:
    - sincrono: via clock;
    - asincrono: via message passing;
  - attraverso dati condivisi: asincrono.

- Geometria delle interconnessioni:
  - irregolari;
  - regolari:
    - albero;
    - shuffle;
    - crossbar;
    - array:
      - monodimensionale;
      - bidimensionale:
        - quadrato;
        - esagonale;
        - triangolare.

- Modulo di granularità:
  - piccolo e costante;
  - piccolo;
  - largo.

Naturalmente non tutte le possibili combinazioni di questi parametri sono significative. I due parametri più utili per caratterizzare i sistemi sistolici sono il controllo della concorrenza ed il modulo di granularità, dove con l'ultimo termine si intende indicare la porzione di elaborazione che un singolo processore può eseguire senza richiedere la comunicazione con un processore vicino, e indica il livello di 'granularità', appunto, in cui una computazione può essere scomposta ed assegnata a processori paralleli. Questi due parametri permettono già di identificare tre classi di architetture parallele particolarmente significative come è mostrato in fig. 1: MIMD (multiple instruction stream - multiple data stream), SIMD (single instruction stream - multiple data stream), Sistolici.

Per quanto riguarda gli altri due parametri, all'interno della classe dei Sistolici si hanno generalmente processori omogenei ed una geometria delle connessioni regolare molto spesso organizzata ad array. Come si è accennato prima un particolare sistema sistolico si presta ad eseguire in maniera efficiente una classe di algoritmi che normalmente è molto ristretta; la ragione principale di ciò risiede nel fatto che negli algoritmi sistolici, dato il loro piccolo modulo di granularità, la comunicazione tra processori assume un ruolo fondamentale nel determinare la complessità di una computazione. È quindi molto importante che il flusso regolare e sincrono di dati e risultati parziali che caratterizza un particolare algoritmo, sia realizzato nel modo più diretto possibile dalla corrispondente architettura. In fig. 2 vengono illustrati degli esempi della corrispondenza che c'è tra geometria delle interconnessioni ed algoritmi dovuta all'esigenza di mappare il "data path" direttamente nella topologia dell'architettura. Il modulo di granularità piccolo fà, inoltre, propendere per l'utilizzo di processori con funzionalità limitate e ben specializzate.

In breve i principi fondamentali su cui si basano i sistemi sistolici possono essere così sintetizzati:

— I dati fluiscono ritmicamente dalla memoria centrale (o da una qualsiasi device esterna) attraverso ogni Processor Element (PE) prima che i risultati tornino in memoria; si ha così un utilizzo multiplo ed esaustivo di ogni dato prelevato dalla memoria: multiplo perchè ogni singolo dato non viene utilizzato solo quando è prelevato in input ma viene riutilizzato da

Tre classi significative

classe	controllo concorrenza	granularità
MIMD	via shared data o message passing, asincrono	larga
SIMD	Centralizzato	piccola costante piccola larga
Sistolici	Distribuito semplice e locale	piccola costante

Fig. 1

Geometrie ed algoritmi

Array lineare	FIR-filter, convoluzione, DFT, moltiplicazione matrice-vettore, on line sort, real-time priority queue, pipeline arithmetic units, ...
Array Quadrato	image processing, chiusura transitiva, moltiplicazione di matrici, algoritmi su grafi, relational data operation, ...
Array Esagonale	operazioni su matrici, chiusura transitiva, DFT, ...
Alberi	algoritmi di ricerca (systolic search tree), parallel function evaluation, recurrence evaluation, ...
Shuffle exchange	Fast Fourier Trasform, bitonic sort, ...

Fig. 2

più processori mentre fluisce nell'array, esaustivo perchè ogni singolo dato prelevato viene restituito in memoria (o scaricato) solo quando tutte le operazioni che necessitano di esso sono state eseguite.

Questo flusso ritmico di dati, tipico dei sistemi *pipelining,* permette di ottimizzare l'utilizzazione delle risorse e in particolare riduce le richieste di I/O facendo si che i dati vengano riutilizzati ed evitando ulteriori accessi in memoria centrale ogni volta che un dato è referenziato. Naturalmente perchè ciò avvenga nell'array di processori sistolici ci deve essere una quantità di memoria (distribuita) sufficiente per trattenere i dati per il tempo necessario alla computazione.

— I dati possono essere:
  - dati di input
  - risultati parziali.

— Il flusso dei dati può avvenire:
  - in più direzioni
  - con velocità diverse.

Si osservi che nei sistemi pipelining classici fluiscono solo i risultati parziali e, normalmente, in una sola direzione.

— Ogni PE esegue operazioni molto semplici.

— Ogni PE è dotato di una quantità di memoria limitata che non dipende dalla dimensione dell'array ma solo dalla natura dell'algoritmo.

— La comunicazione tra i processori è semplice, regolare e locale.

— La comunicazione con il mondo esterno avviene solo attraverso i PE perifierici: solo i processori che si affacciano ai bordi possono servire da ports di I/O per il sistema.

In effetti vedremo che quest'ultima caratteristica è soddisfatta solo dai così detti sistemi "sistolici puri", mentre nei sistemi "semisistolici" i dati possono essere caricati dall'esterno in ciascun PE dell'array.

— L'array di PE può essere esteso in modo modulare per soddisfare richieste di performaces variabili e in particolare per ottenere il bilanciamento di computazione e I/O.

Tipicamente, gli input entrano nell'array attraverso i processori periferici e si propagano ritmicamente ai processori vicini per essere ulteriormente elaborati. Oltre al *pipelining* dei dati si ha una forma di *computational pipelining* in cui l'informazione fluisce da un processore all'altro in un ordine predeterminato. Questa informazione può essere interpretata dal ricevente come dati, come controllo o come una combinazione di entrambi. Ogni output viene calcolato tramite l'esecuzione di una sequenza predeterminata di operazioni elementari compiute in istanti diversi da processori diversi. L'output di un processore è usato come input dal processore vicino ed ogni operazione si svolge dopo che ogni singolo processore ha ricevuto gli opportuni input; la computazione complessiva realizzata dall'array non è però del tipo *data flow.* Infatti le operazioni sono eseguite secondo una sequenza ed una sincronizzazione determinate dallo schema del particolare array sistolico, e non dalla disponibilità dei dati da elaborare; questa disponibilità deve essere assicurata da una attenta scomposizione dell'algoritmo in operazioni elementari e dalla loro sincronizzazione controllata da un clock globale.

In virtù di queste caratteristiche, gli array sistolici si adattano perfettamente alle richieste avanzate dalla tecnologia VLSI: massimizzano i vantaggi offerti in termini di potenza computazionale e, allo stesso tempo, soddisfando naturalmente alle restrizioni da essa imposte, non risentono dei maggiori punti di debolezza che questa tecnologia comporta. Lo studio e lo sviluppo dei sistemi sistolici è, in effetti, uno dei più significativi prodotti del nuovo environment tecnologico costituito dal VLSI. Infatti per sfruttare al meglio le potenzialità offerte dalla tecnologia occorrevano anche idee nuove in termini di metodologia di progettazione, modelli computazionali e architetture di computer.

Vediamo in sintesi quali sono i vantaggi offerti dalla tecnologia VLSI che i sistemi sistolici tendono a massimizzare, e quali le restrizioni a cui naturalmente soddisfano.

Vantaggi

— Grande quantità di Hardware disponibile a basso costo

— Dimensioni fisiche ridotte e conseguente ridotta dissipazione di potenza

— Maggiore affidabilità a livello circuitale dovuta alla minore dissipazione di potenza e alla possibilità di introdurre ridondanza nel circuito.

— L'alto livello di integrazione elimina la necessità di una separazione fisica dei processori della memoria; si ha così la possibilità di superare il bottleneck nella comunicazione memoria processore tramite l'utilizzo di una memoria distribuita.

— La vasta disponibilità di hardware permette di sfruttare in modo effettivo, e ad un basso livello di granularità, il parallelismo e il pipelining tipici degli algoritmi sistolici.

Restrizioni

— Lo sfruttamento efficiente dell'area disponibile comporta una certa limitazione nel numero di I/O pads, questo è in cosiddetto I/O bottleneck tipico dei sistemi VLSI. Si hanno quindi le seguenti restrizioni:
- gli algoritmi devono essere partizionabili in modo da ridurre al massimo la comunicazione tra sottomoduli;
- i dati devono essere usati esaustivamente da ogni modulo VLSI per minimizzare le richieste di I/O.

— La comunicazione tra i gates è costosa quanto i gates stessi ed il tempo di propagazione dei segnali è confrontabile con il tempo di switching; quindi per un utilizzo effettivo delle risorse area e tempo l'hardware deve contenere solo connessioni locali e gli algoritmi devono richiedere solo trasferimenti locali di dati per poter essere mappati efficientemente in un hardware VLSI.

Per quanto riguarda la metodologia di progettazione i sistemi sistolici offrono una via più diretta e veloce per la progettazione dei sottocomponenti a basso livello a partire dalla definizione funzionale del sistema globale (specifiche iniziali). Poiché un array sistolico è costituito da un gran numero di celle appartenenti a pochi tipi differenti, il processo di raffinare il sistema globale e progettare ogni sottocomponente è più facile e più veloce che nei sistemi tradizionali della stessa dimensione ma con celle più differenziate. I costi ridotti di progettazione si possono in sintesi ricondurre ai seguenti punti:

— Progettazione e testing di un numero ridotto di semplici PE base differenti: tutti i PE nel chip sono copie di tali PE base.

— La regolarità delle interconnessioni rende possibile un tipo di progettazione modulare ed estensibile simile a certe tecniche proprie dell'ingegneria del software: si possono progettare chip grossi assemblandone di più piccoli.

— Sfruttando le caratteristiche di modularità, pilening e multiprocessing è possibile soddisfare le richieste di performance di un particolare special purpose chip semplicemente includendo molti PE identici nello stesso chip.

3. ALGORITMI SISTOLICI

Le proprietà peculiari degli algoritmi sistolici sono la regolarità, la ricorsività, la località della computazione. Essi permettono di ottenere throughput molto alti specialmente in applicazioni cosiddette *compute bound.* Con questo termine si intende che il numero totale di operazioni è più grande del numero totale di elementi di input e output. Algoritmi di questo tipo sono utilizzati in varie applicazioni tra cui digital signal processing, image processing, algebra lineare, pattern recognition ... etc. Tali algortimi sono generalmente dominati da tecniche di trasformazione, filtraggi di convoluzione o correlazione e da un limitato numero di tecniche base di algebra lineare. Spesso le applicazioni menzionate si devono svolgere in ambiente real-time e richiedono un alto throughput ed una notevole *processing bandwidth,* possibilmente a costo di un tempo di risposta più alto.

Come primo esempio di applicazione adatta ad un approccio sistolico consideriamo un problema di convoluzione monodimensionale discreta.

Date due sequenze finite di numeri:

$X = (x_0,...,x_{n-1})$ e

$W = (w_0,...,w_{k-1})$
(X: sequenza di input, W: sequenza dei pesi)

si definisce la loro convoluzione lineare aperiodica le sequenza:

$Y = (y_0,...,y_{n+k-2})$ ove gli y_i sono dati dalla formula:

$y_i = \Sigma\, w_j x_{i-j} \quad i = 0,...,n+k-2.$

La convoluzione è un tipico problema compute bound. Infatti, poiché ogni x_i deve essere multiplicato con k pesi w_j ed i risultati parziali addizionati, il numero totale di operazioni è approssimativamente dato da:

n (k *moltiplicazioni* + k-1 *addizioni*) $\approx 2kn$

che è senz'altro maggiore del numero totale di elementi di I/O.

In una architettura convenzionale, con una limitata *memory bandwidth,* gli accessi in memoria sono proporzionali al numero di operazioni; quindi a causa del cosiddetto 'Von Neuman bottleneck' anche una computazione che teoricamente è compute-bound, diviene, in fase di esecuzione, I/O-bound cioè è dominata dall'efficienza delle operazioni di I/O. In una architettura sistolica vengono eseguite operazioni multiple per ogni accesso in memoria, è quindi possibile incrementare l'efficienza delle computazioni compute-bound senza richiedere aumenti di *memory bandwidth.*

L'altra caratteristica principale che fa propendere per un approccio sistolico nel caso della convoluzione è la regolarità: uno stesso tipo di operazione, il prodotto interno, è

eseguita su ogni dato in modo regolare e ripetitivo. Ciò ben si adatta alla regolarità delle connessioni e all'uniformità dei processori tipiche delle architetture sistoliche.

Oltre che per le due caratteristiche menzionate sopra la convoluzione viene solitamente scelta come esempio paradigmatico perchè è semplice, ha una larga applicazione in DSP ed è rappresentativa di una vasta classe di problemi; gli algoritmi di convoluzione si possono infatti adattare per risolvere problemi di:

— FIR (Finite Impulse Response)

— correlazione (backward convolution)

— pattern matching (sostituendo il *prodotto* con il *confronto* e il + con l'*and logico*)

— moltiplicazione di numeri binari (usando il *prodotto di bit* e la *somma con carry*)

— signal resampling

— language recognition

— recursive filtering

— DFT

Il modo più ovvio per rendere possibile l'uso multiplo di ogni elemento di input è quello di fare ricorso ad un dispositivo di comunicazione globale per realizzare il broadcasting degli inputs. Il global broadcasting unitamente ad un'opportuna scomposizione della computazione e alle consuete tecniche di pipelining dei risultati parziali e/o dei pesi permette di risolvere l'I/O bottleneck tipico dei sistemi VLSI con un approccio cosiddetto semisistolico; illustreremo prima questo tipo di approccio per le minori difficoltà di temporizzazione che presenta [1].

Gli inputs x_i sono trasmessi, contemporaneamente, ad ogni processore con global broadcasting, uno per ogni ciclo di clock. I pesi w_i sono caricati inizialmente (preloaded), uno in ogni processore (Fig. 3). I risultati parziali y_i si muovono sistolicamente lungo l'array lineare: sono dapprima inizializzati a zero e quindi, attraversando l'array, vengono modificati in accordo all'equazione ricorsiva scritta in figura. Dopo ogni colpo di clock il risultato parziale calcolato da un processore si propaga al processore successivo. Abbiamo supposto che ogni processore sia in grado di eseguire una moltiplicazione e una addizione in un solo colpo di clock; questa assunzione può sempre essere soddisfatta se ci si riferisce ad un clock esterno "di sincronizzazione" e si permette che ogni processore lavori sulla base di un clock interno (anch'esso globale) di frequenza più alta. Si può anche immaginare che ogni processore esegua le sue operazioni in modo "pipelining" sincronizzato dal clock interno, così da raggiungere throughputs ancora più alti; in questi casi si parla di "two-level pipelined sistolic array". Per appoggiare l'intuizione di come si svolge la computazione nell'architettura mostrata prima è utile il diagramma di temporizzazione in fig. 4 che mostra in quale ciclo viene prodotto ogni risultato parziale.

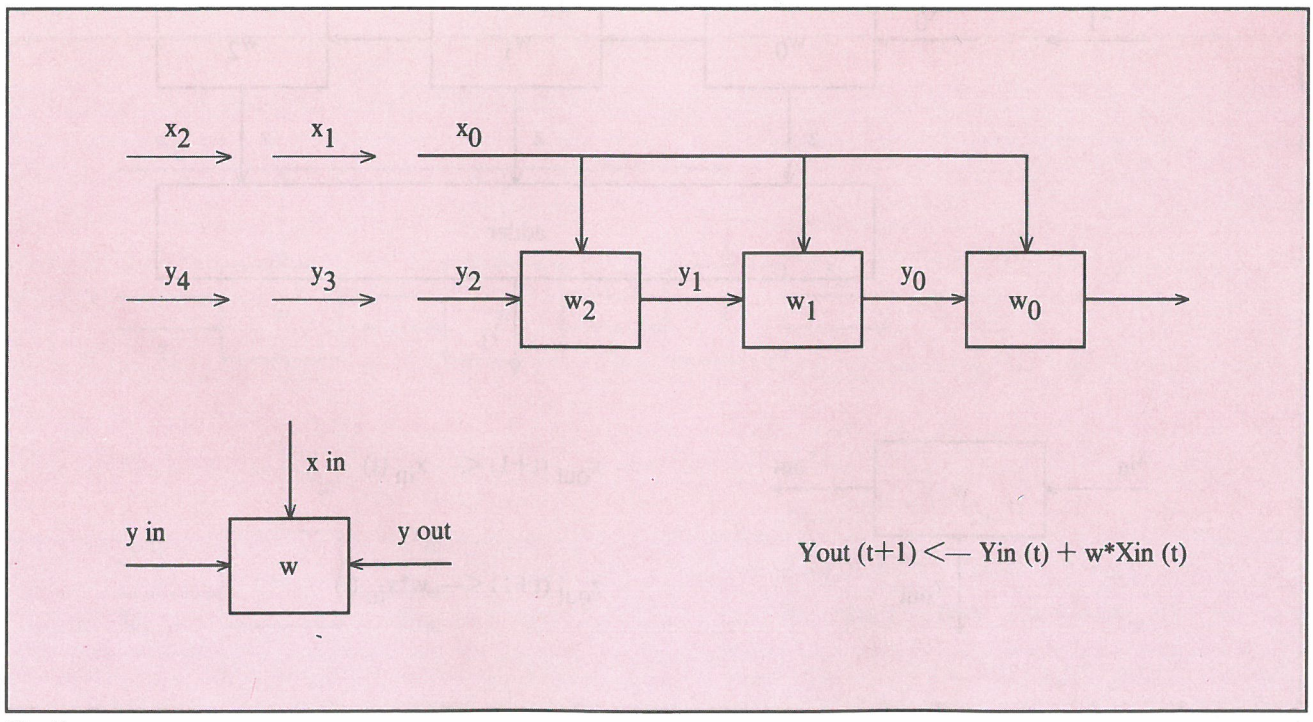

Fig. 3

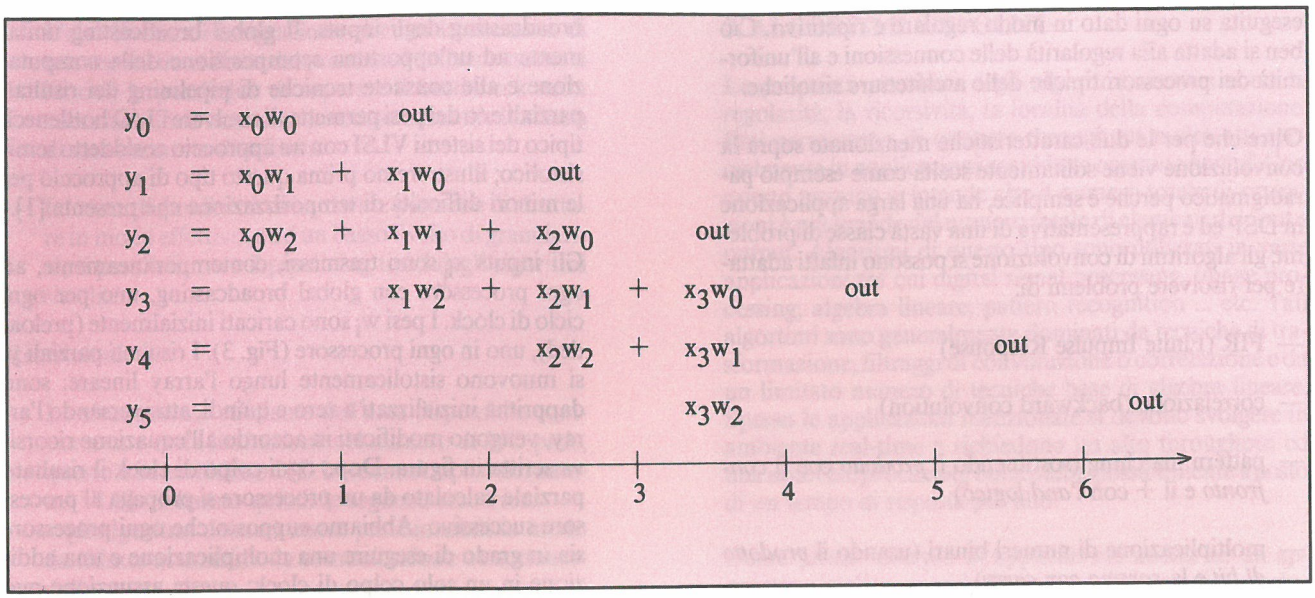

Fig. 4

Un'altra tecnica che fa uso di schemi di comunicazione non locali per superare l'I/O bottleneck è detta *unbounded fan-in;* essa è l'opposto del broadcasting in quanto consiste nel raccogliere i dati contemporaneamente da un numero (non necessariamente limitato) di processori attraverso una rete o un bus di comunicazione globale.

Vediamo uno schema semi-sistolico di convoluzione basata su questa tecnica [1] (Fig. 5).

Questo schema si può sintetizzare così:

— gli input x_i si propagano sistolicamente da un processore all'altro ad ogni colpo di clock;

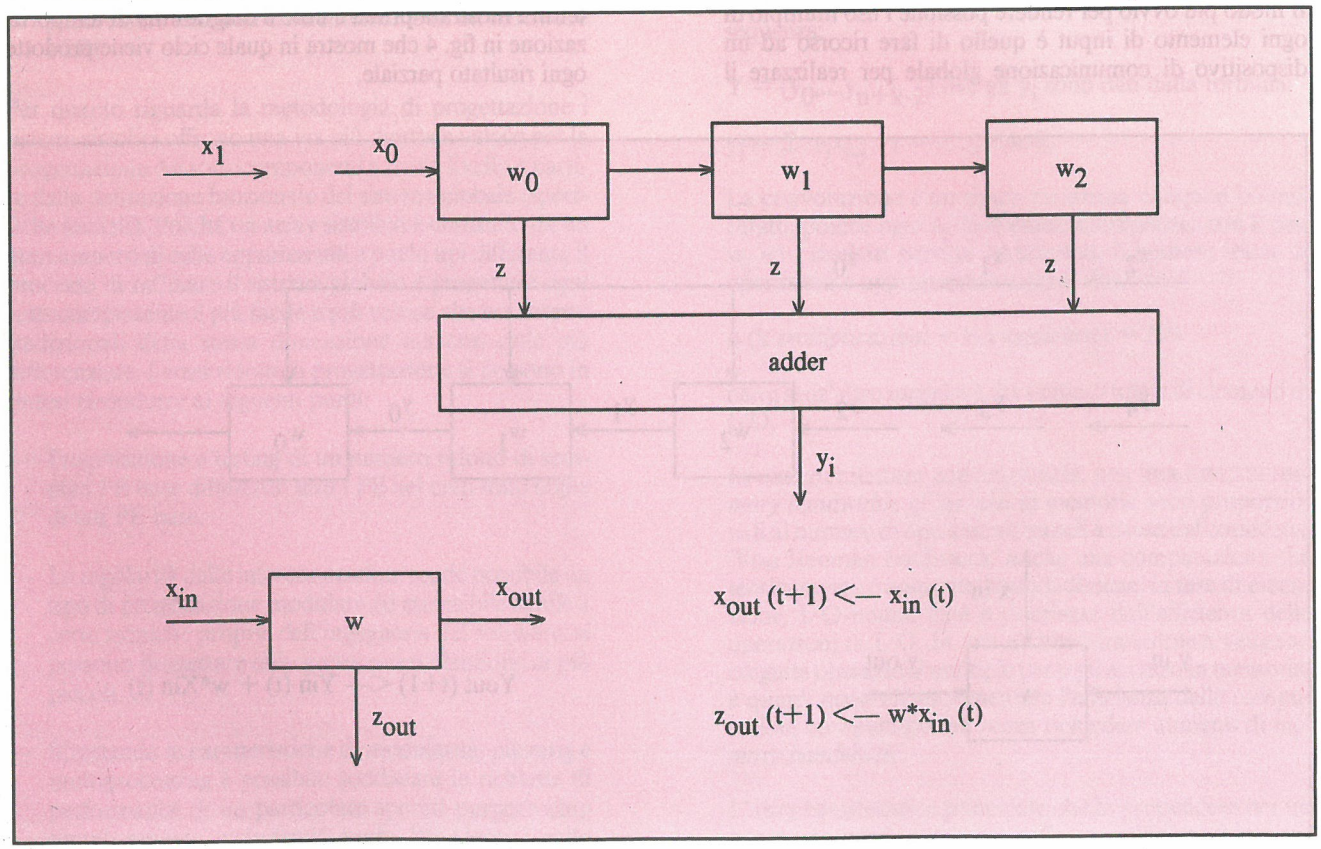

Fig. 5

— i pesi w_i sono già caricati nei processori;

— fan-in degli output che vengono raccolti e addizionati ad ogni colpo di clock.

Per evitare ritardi ad ogni ciclo l'adder può essere implementato mediante un *pipelined adder tree.*

Il relativo diagramma di temporizzazione è dato in fig. 6.

Un inconveniente presentato dai due algoritmi presi in considerazione è il tempo di preloading che non può essere ignorato se i pesi cambiano da convoluzione a convoluzione; in questo caso infatti non si può più avere tempo di risposta costante.

Per ovviare a questo inconveniente si può fare in modo che i pesi vengano caricati a run time; ad esempio si possono fare scorrere gli input x_i sistolicamente lungo l'array in direzione opposta ai risultati parziali, e trasmettere i pesi w_i mediante global broadcasting, uno ad ogni colpo di clock (Fig. 7). Affinché ogni y_i incontri tutti gli x_i che compaiono nella sua sommatoria sia gli y_i che gli x_i dovranno scorrere in processori alterni; ciò equivale a porre dei "placeholders", che verranno ignorati dai PE, tra ogni successivo elemento delle sequenze y e x. Inoltre supponiamo che un tag si propaghi insieme a x_0, il suo scopo è quello di congelare il peso correntemente trasmesso nel processore opportuno che è appunto quello in cui di volta in volta è presente il tag.

Se si osserva il diagramma in fig. 8 si può notare che l'output viene prodotto ogni due cicli di clock anziché ogni ciclo come in fig. 4. Questo è il prezzo che si è pagato per eliminare il tempo di preloading. Si noti però che ad ogni ciclo compiono operazioni utili solo processori alterni; ciò fa sì che il throughput potenziale dell'array sia doppio: con lievi modifiche dei processori, come l'aggiunta di un accumulatore, si possono calcolare due correlazioni intercalate nello stesso tempo.

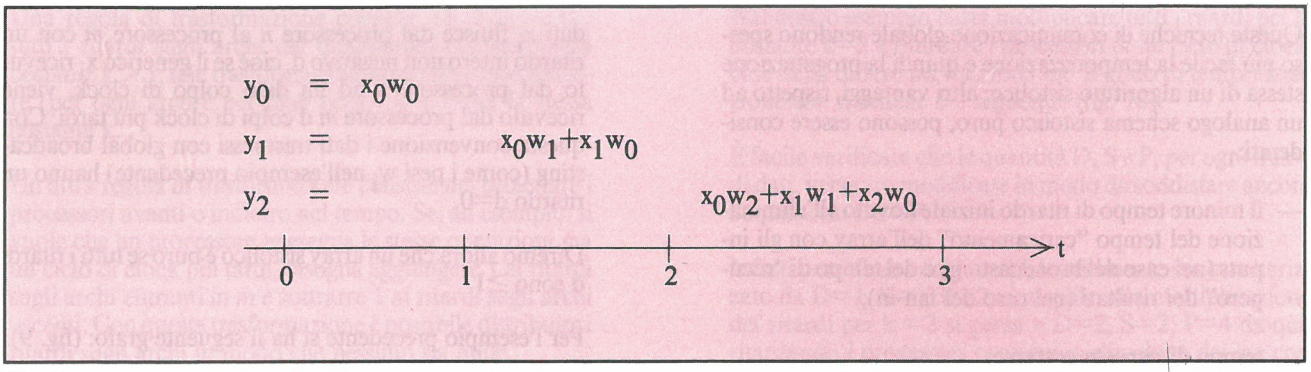

Fig. 6

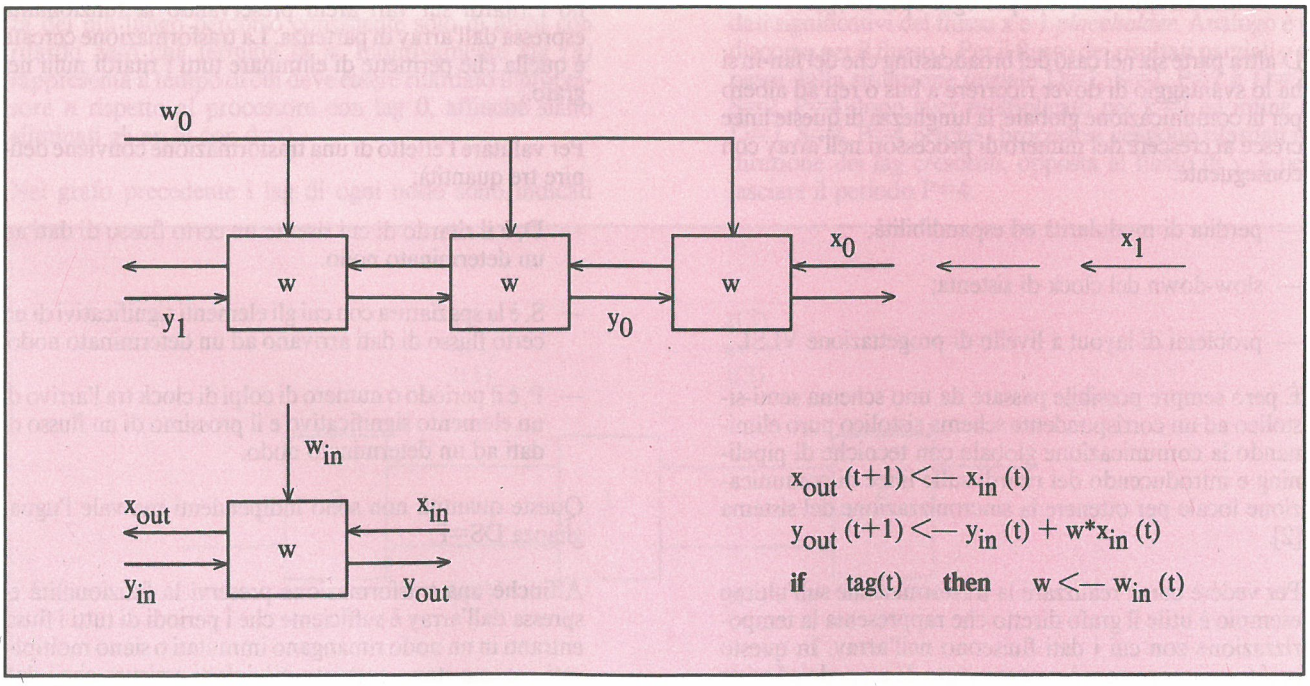

Fig. 7

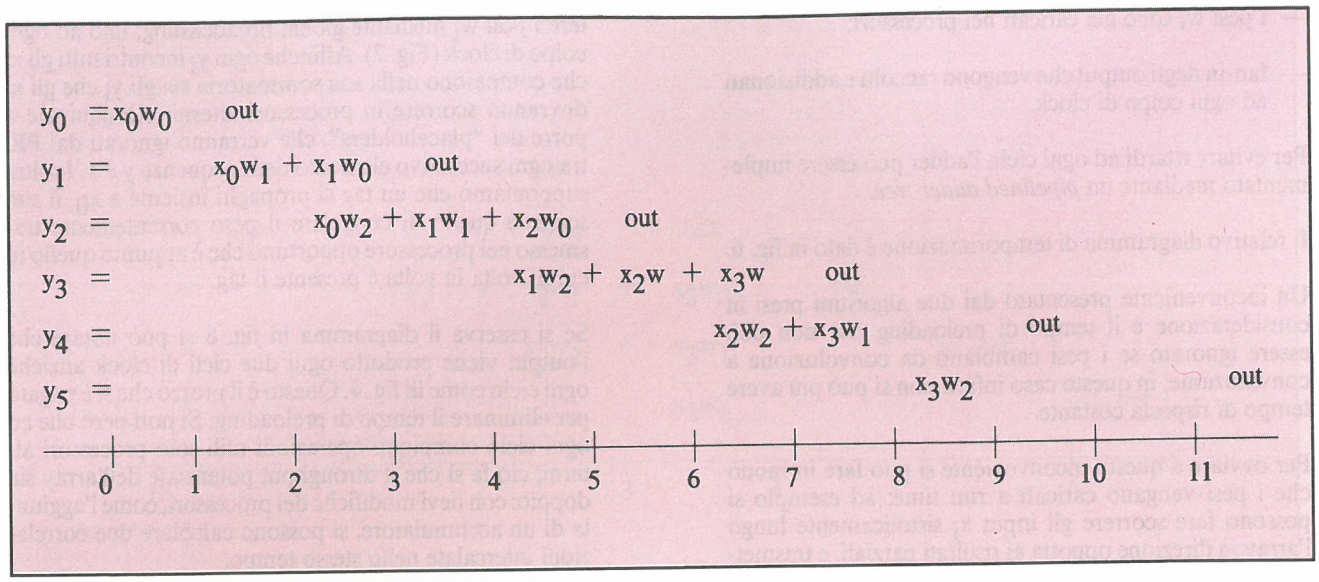

Fig. 8

Queste tecniche di comunicazione globale rendono spesso più facile la temporizzazione e quindi la progettazione stessa di un algoritmo sistolico; altri vantaggi, rispetto ad un analogo schema sistolico puro, possono essere considerati:

— il minore tempo di ritardo iniziale dovuto all'eliminazione del tempo "caricamento" dell'array con gli inputs (nel caso del broadcasting) o del tempo di "recupero" dei risultati (nel caso del fan-in);

— tempo di risposta costante;

— minore numero di cicli per ogni operazione.

D'altra parte sia nel caso del broadcasting che del fan-in si ha lo svantaggio di dover ricorrere a bus o reti ad albero per la comunicazione globale; la lunghezza di queste linee cresce al crescere del numero di processori nell'array con conseguente:

— perdita di modularità ed espandibilità;

— slow-down del clock di sistema;

— problemi di layout a livello di progettazione VLSI.

È però sempre possibile passare da uno schema semi-sistolico ad un corrispondente schema sistolico puro eliminando la comunicazione globale con tecniche di pipelining e introducendo dei ritardi sulle linee di comunicazione locale per ottenere la sincronizzazione del sistema [2].

Per vedere come realizzare la trasformazione sull'ultimo esempio è utile il grafo diretto che rappresenta la temporizzazione con cui i dati fluiscono nell'array. In questo grafo ogni processore è rappresentato da un nodo ed esiste un arco, labellato x(d), tra i nodi *n* e *m* se la sequenza dei dati x_i fluisce dal processore *n* al processore *m* con un ritardo intero non negativo d, cioè se il generico x_i ricevuto dal processore *n* ad un dato colpo di clock, viene ricevuto dal processore *m* d colpi di clock più tardi. Con questa convenzione i dati trasmessi con global broadcasting (come i pesi w_i nell'esempio precedente) hanno un ritardo d=0.

Diremo allora che un array sistolico è puro se tutti i ritardi d sono ≥ 1.

Per l'esempio precedente si ha il seguente grafo: (fig. 9).

Stabiliamo delle regole di trasformazione che modifichino i ritardi sui vari archi preservando la funzionalità espressa dall'array di partenza. La trasformazione cercata è quella che permette di eliminare tutti i ritardi nulli nel grafo.

Per valutare l'effetto di una trasformazione conviene definire tre quantità:

— D, è il ritardo di cui risente un certo flusso di dati ad un determinato nodo.

— S, è la spaziatura con cui gli elementi significativi di un certo flusso di dati arrivano ad un determinato nodo.

— P, è il periodo o numero di colpi di clock tra l'arrivo di un elemento significativo e il prossimo di un flusso di dati ad un determinato nodo.

Queste quantità non sono indipendenti ma vale l'uguaglianza DS=P.

Affinché una trasformazione preservi la funzionalità espressa dall'array è sufficiente che i periodi di tutti i flussi entranti in un nodo rimangano immutati o siano moltiplicati per una stessa costante; ciò infatti assicura che i dati necessari ad una determinata operazione giungano ancora

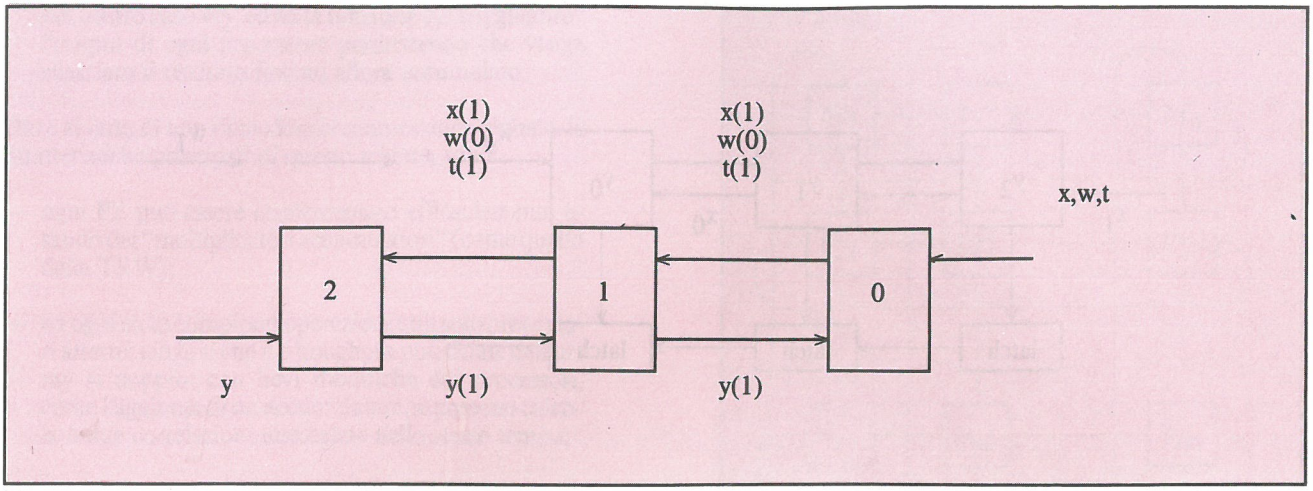

Fig. 9

"simultaneamente" al processore opportuno.

Una regola di trasformazione consiste nel moltiplicare tutti i ritardi sugli archi del diagramma per una stessa costante $k > 1$; tale trasformazione equivale a moltiplicare (per ogni arco e per ogni nodo) D e P per la stessa costante k.

Un'altra regola di trasformazione consiste nel proiettare i processori avanti o indietro nel tempo. Se, ad esempio, si vuole che un processore *m* esegua le stesse operazioni ma un ciclo di clock più tardi, bisogna aggiungere 1 ai ritardi sugli archi entranti in *m* e sottrarre 1 ai ritardi sugli archi uscenti. Con questa trasformazione è possibile distribuire i ritardi sugli archi in modo che nessuno sia nullo.

Per ogni singolo nodo *n* definiamo con lag *(n)* la lunghezza del più lungo percorso consistente solo di archi con ritardo zero che termina in *n*. Intuitivamente lag *(n)* rappresenta il tempo di cui deve essere ritardato il processore *n* rispetto al processore con lag 0, affinché siano eliminati gli archi con d=0.

Nel grafo precedente i lag di ogni nodo sono indicati all'interno dei rettangoli.

Nel nostro esempio basta moltiplicare tutti i ritardi per la costante k=2 e ritardare i processori di un ciclo di clock, cioè della differenza tra i loro lag. Il grafo di temporizzazione che si ottiene è il seguente: (fig. 10).

È facile verificare che le quantità D, S e P, per ogni flusso di dati, vengono modificate in modo da soddisfare ancora l'equazione DS=P.

Infatti per il flusso degli x, che inizialmente era caratterizzato da D=1, S=2, P=2, in seguito alla moltiplicazione dei ritardi per k =2 si passa a D=2, S=2, P=4 da qui, ritardando i processori contigui in direzione dei lag crescenti si ha D=3 e quindi, poiché P deve rimanere costante, S=4/3. Ciò significa che ogni 4 posizioni ci saranno 3 dati significativi del flusso x e 1 *placeholder*. Analogo è il discorso per il flusso t. Per il flusso dei risultati parziali y si passa dalla situazione iniziale D=1, S=2, P=2 a D=2, S=2, P=4 dopo aver moltiplicato per k=2 ed infine a D=1, S=4, P=4 perché i processori vengono ritardati in direzione dei lag crescenti, opposta al flusso di y, e per lasciare il periodo P=4.

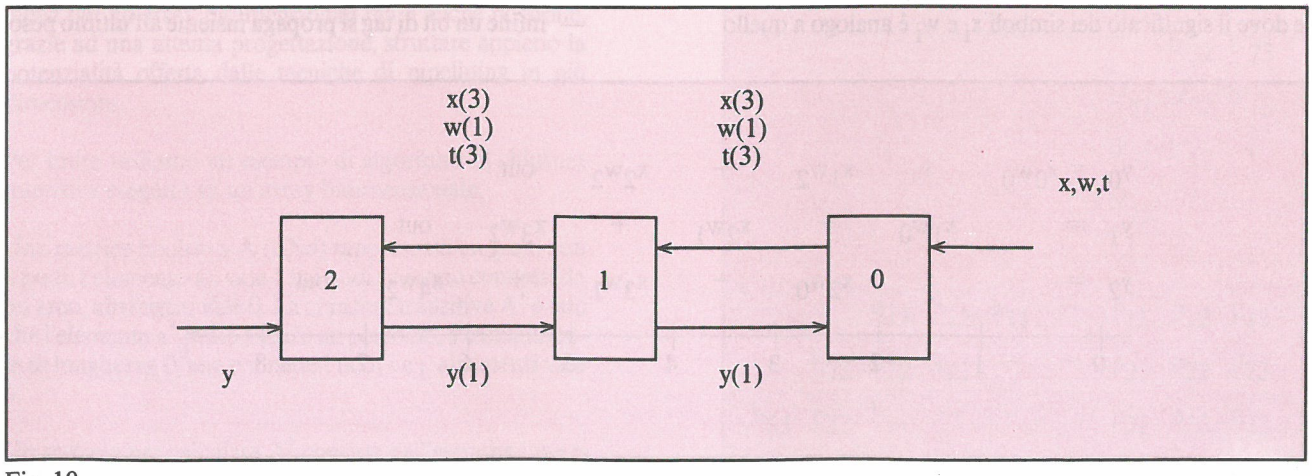

Fig. 10

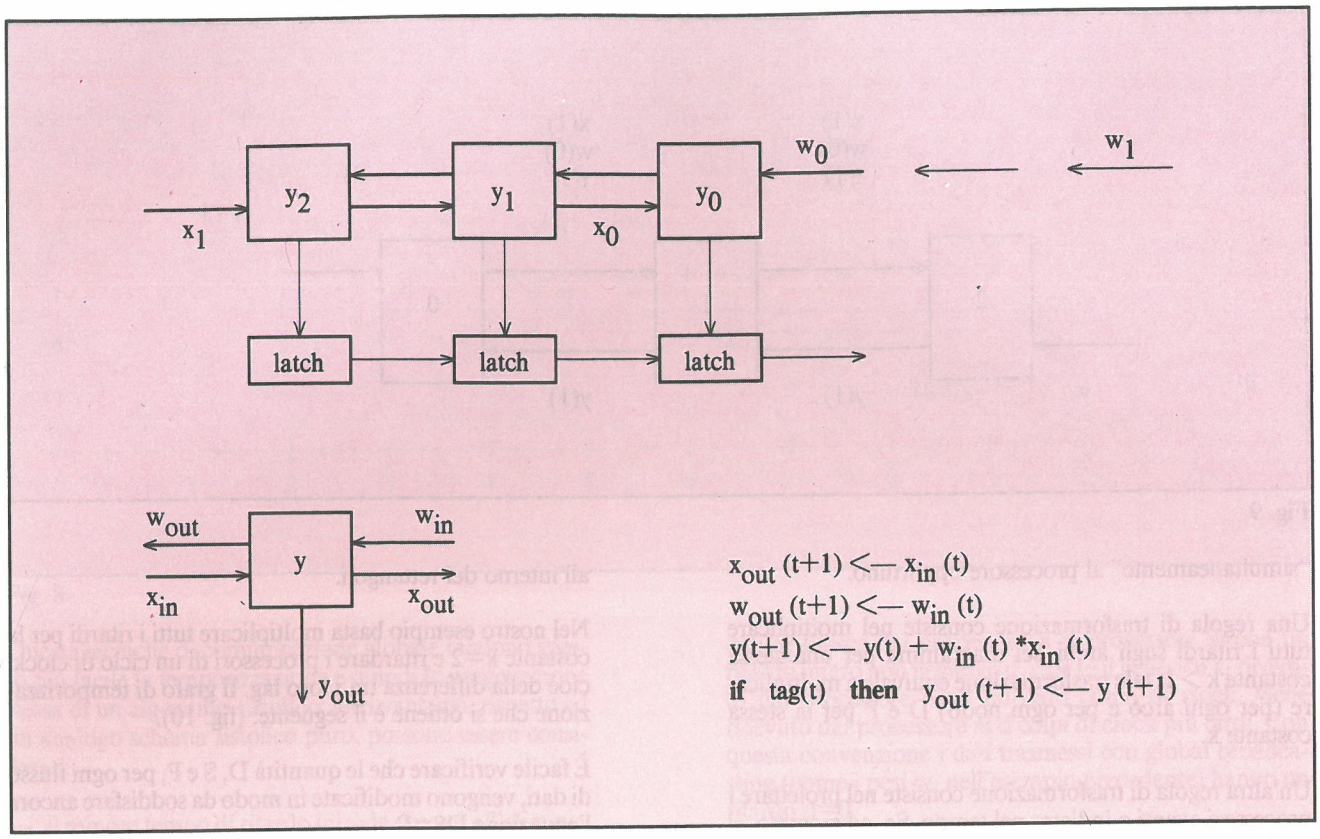

Fig. 11

Infine per il flusso w la situazione è la seguente: inizialmente si ha D=0, P=1 e di conseguenza S è infinito, dopo la moltiplicazione per k si ha P=2, D=0 e S infinito, ritardando i processori si ottiene D=1 (eliminando così il global broadcasting) e, per compensazione S=2 mentre P rimane 2.

Vediamo ora degli schemi sistolici puri per un caso particolare di convoluzione, la correlazione, definita dalla seguente equazione:

$y_i = \Sigma\, w_j x_{i+j}$ con k=3,

e dove il significato dei simboli x_i e w_i è analogo a quello visto prima, mentre gli y_i rappresentano la correlazione di una sequenza di input con una determinata sequenza di pesi [1] (Fig. 11).

— Gli input x_i si muovono sistolicamente in processori alterni;

— i pesi w_i si muovono sistolicamente in processori alterni e in direzioni opposte;

— i risultati parziali y_i sono accumulati in ogni processore e processati in parallelo;

— infine un bit di tag si propaga insieme all'ultimo peso,

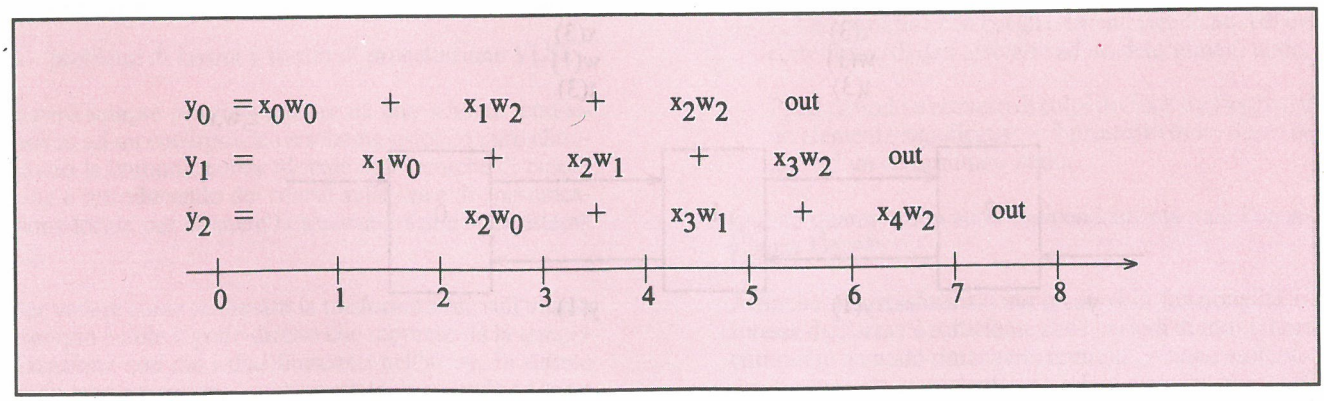

Fig. 12

nel nostro caso w_2, ed ha la funzione di "triggherare" l'output di ogni processore permettendo che venga rilasciato il risultato fino ad allora accumulato.

Oltre al fatto di non richiedere comunicazione globale, le caratteristiche principali di questo schema sono:

— ogni PE può essere implementato efficaciemente usando dei "moltiplicatori accumulatori" (come quello della TRW);

— ad ogni ciclo compiono operazioni utili solo processori alterni; ciò fa si che il trhoughput potenziale dell'array si doppio: con lievi modifiche dei processori, come l'aggiunta di un accumulatore, si possono calcolare due correlazioni intercalate nello stesso tempo;

— l'array presenta un tempo di start up di 2k cicli, come si può vedere dal diagramma di temporizzazione in fig. 12.

Altri schemi di convoluzione sistolica sono riportati in letteratura, ad esempio in [1], [2].

La disponibilità di algoritmi sistolici di convoluzione caratterizzati da un alto throughput apre la via ad un nuovo approccio ad uno dei più classici problemi di DSP: il calcolo della trasformata di Fourier. Una volta il metodo più efficiente per calcolare una convoluzione era quello di trasformare il problema di convoluzione nel problema del calcolo di una trasformata di Fourier, avvalendosi del ben noto teorema di convoluzione in base al quale la trasformata di Fourier Y della convoluzione y (periodica) di due sequenze (periodiche) x e w, è uguale al prodotto delle loro trasformate X e W: $Y_k = X_k W_k$ per la k-esima componente. Infatti l'esistenza di algoritmi molto efficienti di FFT faceva sì che fosse conveniente percorrere questa "via lunga" alla convoluzione. Recentemente alcuni autori hanno utilizzato il teorema di convoluzione in senso inverso, cioè hanno impiegato algoritmi sistolici di convoluzione per calcolare efficientemente delle DFT.

Finora abbiamo visto array sistolici lineari; i problemi di temporizzazione e sincronizzazione diventano più complessi per gli array bidimensionali ma è anche possibile, grazie ad una attenta progettazione, sfruttare appieno la potenzialità offerta dalle tecniche di pipelining in più dimensioni.

Per finire vediamo un esempio di algoritmo di chiusura transitiva eseguito su un array bidimensionale.

Una matrice booleana A (nXn) rappresenta un grafo con n nodi: l'elemento a_{ij} vale 1 se i nodi i e j sono connessi da un arco, altrimenti vale 0. La chiusura transitiva A^+ è tale che l'elemento a^+_{ij} vale 1 se c'è un percorso, eventualmente di lunghezza 0, che connette i nodi i e j, altrimenti vale 0.

Un metodo per calcolare A^+ consiste nell'eseguire $\log_2 n$ moltiplicazioni booleane (con l'and e l'or logici al posto

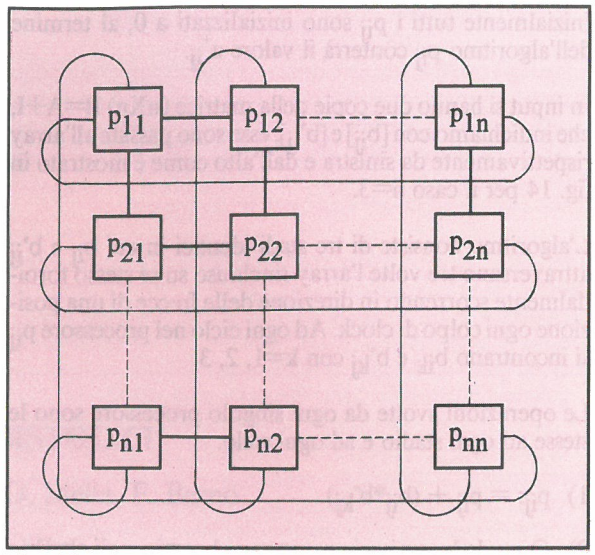

Fig. 13

del prodotto e dell'addizione) della matrice A+I; usando i migliori algoritmi di moltiplicazione di matrici, ad esempio degli algoritmi sistolici, questo metodo comporta una complessità:

— in tempo $O(n\log_2 n)$

— in area $O(n^2)$.

Il seguente algoritmo sistolico di chiusura transitiva, sviluppato da Guibas, Kung e Thompson, permette di ridurre la complessità in tempo a O(n). [2].

Lo schema si basa su di un array bidimensionale di processori con connessioni a rete quadrata rinchiuso ai bordi in modo toroidale come è rappresentato in fig. 13.

Con p_{ij} indichiamo sia il processore che il valore booleano che contiene.

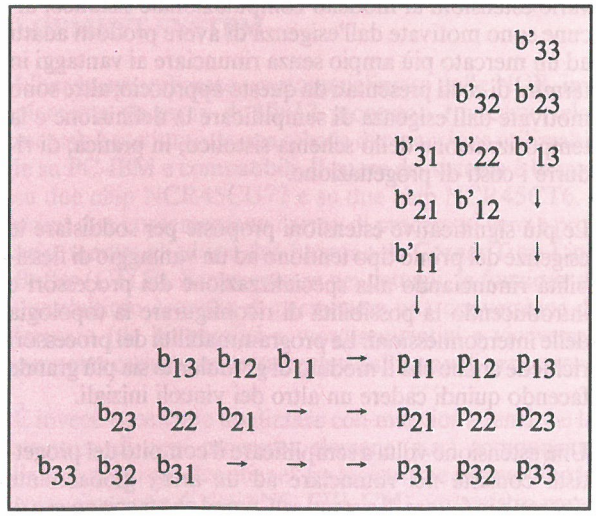

Fig. 14

Inizialmente tutti i p_{ij} sono inizializzati a 0, al termine dell'algoritmo p_{ij} conterrà il valore a^{+}_{ij}.

In input si hanno due copie della matrice (nXn) B=A+I, che indichiamo con $[b_{ij}]$ e $[b'_{ij}]$; esse sono passate all'array rispettivamente da sinistra e dall'alto come è mostrato in fig. 14 per il caso n=3.

L'algoritmo consiste di tre stadi identici in cui b_{ij} e b'_{ij} attraversano tre volte l'array rinchiuso su se stesso toroidalmente scorrendo in direzione delle frecce di una posizione ogni colpo di clock. Ad ogni ciclo nel processore p_{ij} si incontrano b_{ik} e b'_{kj} con k=1, 2, 3.

Le operazioni svolte da ogni singolo processore sono le stesse ad ogni stadio e ad ogni ciclo:

1) $p_{ij} := p_{ij} + (b_{ij} * b'_{kj})$

2) Quando b_{ij} arriva in p_{ij} assegno $b_{ij} := p_{ij}$; gli altri b_{ik} con $k \neq j$ non vengono modificati da p_{ij}. Analogamente per b'_{ij} si ha $b'_{ij} := p_{ij}$, mentre i b_{kj} con $k \neq i$ non vengono modificati da p_{ij}.

Osserviamo che, considerando il ciclo 1 quello in cui b_{ij} e b'_{ij} arrivano in p_{11}, b_{ij} arriva in p_{ij} al ciclo i+2j-2, mentre b'_{ij} arriva in p_{ij} al ciclo 2i+j-2. Quindi per avvisare p_{ij} dell'arrivo di b_{ij} è sufficiente che un segnale originato in p_{11} si propaghi duplicandosi verso il basso ogni ciclo e verso destra ogni due cicli. La prima volta che una copia di questo segnale raggiunge p_{ij} indica l'arrivo di b_{ij}. La stessa argomentazione si applica a b'_{ij} scambiando "destra" con "basso". Al termine del terzo stadio ogni p_{ij} conterrà a^{+}_{ij} cioè il corrispondente elemento della matrice A^{+} cercata [2].

4. CONCLUSIONI

Nel corso dell'articolo abbiamo posto l'accento su alcuni vincoli intesi a definire lo spazio delle architetture sistoliche in senso classico. Recentemente sono state sviluppate varie estensioni al modello computazionale sistolico; alcune sono motivate dall'esigenza di avere prodotti adatti ad un mercato più ampio senza rinunciare ai vantaggi in termini di costi presentati da questo approccio; altre sono motivate dall'esigenza di semplificare la definizione e la temporizzazione dello schema sistolico, in pratica, di ridurre i costi di progettazione.

Le più significative estensioni proposte per soddisfare le esigenze del primo tipo tendono ad un vantaggio di flessibilità rinunciando alla specializzazione dei processori e introducendo la possibilità di riconfigurare la topologia delle interconnessioni. La programmabilità dei processori richiede anche che il modulo di granularità sia più grande facendo quindi cadere un altro dei vincoli iniziali.

Una estensione volta a semplificare il compito del progettista consiste nel rinunciare ad un array globalmente sincrono in favore di sistemi self-timed che facciano uso di meccanismi di handshaking asincroni tra processori vicini. I sistemi che ne risultano sono talvolta chiamati *Wavefront Arrays,* essi liberano il progettista dal compito di sincronizzare l'array permettendogli di sfruttare il principio della "dataflow computation" per ottenere una più facile estrazione del parallelismo dal problema. Per una rassegna dettagliata su queste estensioni rimandiamo a [3].

5. BIBLIOGRAFIA

[1] H.T. Kung, WHY SYSTOLIC ARCHITECTURES, Computer Magazine 15(1), Ja. 1982

[2] J. Ullman, COMPUTATIONAL ASPECTS OF VLSI, Computer Science Press, cap. 5, Systolic algorithms, 1984

[3] Computer IEEE, NUMERO SPECIALE SUI SISTOLICI, 20(7), July 1987

UN DRIVER PER SISTOLICI

G. Mazzocchi, S. Missaglia, D. Mella, F. Berno

Dipartimento di Scienze dell'Informazione

Università di Milano

RIASSUNTO

Il calcolo parallelo, sinora riservato a calcolatori di grosse dimensioni, ha bisogno di diventare accessibile ad una fascia più larga di utenti. Un buon veicolo di diffusione della conoscenza è il personal computer.
In questo articolo esaminiamo nel dettaglio un particolare sistema di sviluppo dotato di processori sistolici, e presentiamo uno strumento software che ne ha permesso l'integrazione con il tradizionale mondo MS-DOS.

ABSTRACT

Parallel computation, until now reserved to main frames and minicomputers, needs to become accessible to a larger number of users. Personal Computer is a good mean of knowledge diffusion.
In this paper we describe in detail a particular development system equipped with systolic processor, and we present a software tool allowing its integration with traditional MS-DOS world.

1. INTRODUZIONE

L'approccio sistolico alla computazione si basa su uno schema di elaborazione SIMD. In altre parole, ogni processore esegue la stessa istruzione, uguale per tutti i componenti dell'architettura considerata, su dati locali più o meno correlati con quelli dei processori vicini. La potenza di questo modo di calcolare si basa, allora, essenzialmente su due fattori: il parallelismo del calcolo e la capacità di comunicazione fra processori.

In generale, ogni processore comunica con i suoi vicini. La relazione di vicinanza è, ovviamente, dipendente dalla topologia architetturale: i problemi risolvibili con algoritmi sistolici richiedono generalmente architetture diverse tra loro.

Il nostro lavoro è stato svolto su un'architettura quadrata (un array di 12x12 processori) che risulta effiecientemente utilizzabile per una particolare classe di problemi (signal processing, image processing).

In questo articolo presentiamo una introduzione al GAPP NCR (Geometric Arithmetic Parallel Processor) ed una breve rassegna di quanto è stato sviluppato nel laboratorio del SiLab.

2. L'AMBIENTE HARDWARE: IL GAPP DEVELOPMENT SYSTEM

L'hardware utilizzato, commercializzato dalla NCR, implementa un array di 12x12 processori (processor elements); è costituito da una scheda di espansione utilizzabile su PC-IBM e compatibili. Il cuore del sistema è basato su due chip NCR45CG72 e su due chip NCR45CT6. I primi due implementano l'array di processori vero e proprio, mentre gli ultimi due fungono da Corner Turn Line Buffer (CTLB). Senza entrare nei dettagli, la funzione di quest'ultimo è quella di permettere una conversione di formato (da bit-parallel a word-sequential e viceversa) per poter caricare ed estrarre dati dall'array di processori.

È invece necessario analizzare con maggior attenzione la struttura di ogni processor element (p.e.) componente l'array. Ogni p.e. "contiene" tre registri che permettono la comunicazione di dati (NS, EW, CM), un registro carry (C), una semplicissima ALU e 128 bit di RAM. Comin-

ciamo ad esaminare i registri di comunicazione.

I due registri NS (North-South) ed EW (East-West) permettono di scambiare dati con i 4 processori vicini lungo canali di comunicazione bidirezionali, mentre il terzo (CM: CoMmunication) permette un flusso di dati solo sulla direzione da Sud a Nord. Inoltre NS ed EW forniscono i bit di input all'ALU per le operazioni.

Il registro C funge da carry durante le operazioni aritmetico-logiche, ed è l'unico, oltre a CM, che permette di scrivere in memoria. L'ALU fornisce ad ogni istante il risultato della somma e della differenza di EW ed NS, il carry ed il borrow generati durante le operazioni. In ogni operazione C è considerato come carry/borrow precedente.

Una importante caratteristica del GAPP è la possibilità di overlapping di istruzioni: mentre si effettuano trasferimenti di dati è possibile operare in memoria ed ottenere risultati di operazioni aritmetico-logiche. Una più estesa rassegna della potenzialità software del GAPP verrà presentata poco oltre.

È invece interessante capire le modalità di interfacciamento tra PC e scheda GAPP.

L'interfacciamento con l'host avviene grazie all'I/O bus del PC. In particolare ogni registro della scheda è mappato su una porta di I/O del PC-IBM. In totale vengono utilizzate 8 porte di I/O (PO,..,P7) di cui 3 (P5,P6,P7) servono per la sincronizzazione delle operazioni sulla scheda, mentre le altre 5 servono per caricare ed estrarre dati e per eseguire le istruzioni.

Il risultato è, purtroppo, un compromesso poco felice tra

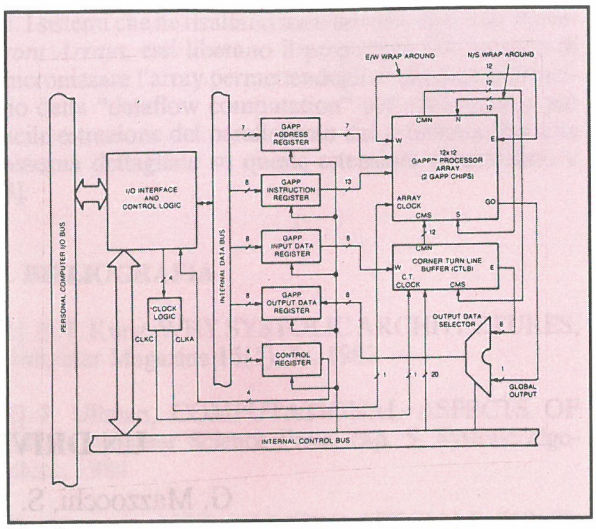

Fig. 1

facilità di programmazione, costi di implementazione e velocità di esecuzione dei programmi. Infatti, fornendo il clock via software, si utilizza un chip che può funzionare con clock fino a 10 MHz a frequenze dell'ordine dei pochi kilohertz.

L'Assembler GAPP ed il GAL

Approfondiamo ora le potenzialità software del GAPP. Il set di istruzioni primitive dell'assembler del GAPP, presentato in figura 2, è abbastanza ridotto; nonostante ciò vedremo come sia più che sufficiente per applicazioni anche non banali. Infatti, la possibilità di overlapping di istruzioni ci permette di generare circa 6000 istruzioni

Register Operation	Mnemonic	Control Lines C_C	C_B	C_A	C_9	C_8	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	Description
CM	CM: = CM	X	X	X	X	X	X	X	X	X	X	X	0	0	MICRO-NOP
	CM: = RAM	X	X	X	X	X	X	X	X	X	X	X	0	1	LOAD CM FROM RAM
	CM: = CMS	X	X	X	X	X	X	X	X	X	X	X	1	0	MOVE FROM CMS INTO CM
	CM: = 0	X	X	X	X	X	X	X	X	X	X	X	1	1	LOAD 0 INTO CM
NS	NS: = NS	X	X	X	X	X	X	X	X	0	0	0	X	X	MICRO-NOP
	NS: = RAM	X	X	X	X	X	X	X	X	0	0	1	X	X	LOAD NS FROM RAM
	NS: = N	X	X	X	X	X	X	X	X	0	1	0	X	X	MOVE FROM N INTO NS
	NS: = S	X	X	X	X	X	X	X	X	0	1	1	X	X	MOVE FROM S INTO NS
	NS: = EW	X	X	X	X	X	X	X	X	1	0	0	X	X	MOVE FROM EW INTO NS
	NS: = C	X	X	X	X	X	X	X	X	1	0	1	X	X	MOVE FROM C INTO NS
	NS: = 0	X	X	X	X	X	X	X	X	1	1	0	X	X	LOAD 0 INTO NS
EW	EW: = EW	X	X	X	X	X	0	0	0	X	X	X	X	X	MICRO-NOP
	EW: = RAM	X	X	X	X	X	0	0	1	X	X	X	X	X	LOAD EW FROM RAM
	EW: = E	X	X	X	X	X	0	1	0	X	X	X	X	X	MOVE FROM E INTO EW
	EW: = W	X	X	X	X	X	0	1	1	X	X	X	X	X	MOVE FROM W INTO EW
	EW: = NS	X	X	X	X	X	1	0	0	X	X	X	X	X	MOVE FROM NS INTO EW
	EW: = C	X	X	X	X	X	1	0	1	X	X	X	X	X	MOVE FROM C INTO EW
	EW: = 0	X	X	X	X	X	1	1	0	X	X	X	X	X	LOAD 0 INTO EW
C	C: = C	X	X	0	0	0	X	X	X	X	X	X	X	X	MICRO-NOP
	C: = RAM	X	X	0	0	1	X	X	X	X	X	X	X	X	LOAD C FROM RAM
	C: = NS	X	X	0	1	0	X	X	X	X	X	X	X	X	MOVE FROM NS INTO C
	C: = EW	X	X	0	1	1	X	X	X	X	X	X	X	X	MOVE FROM EW INTO C
	C: = CY	X	X	1	0	0	X	X	X	X	X	X	X	X	LOAD C FROM CARRY
	C: = BW	X	X	1	0	1	X	X	X	X	X	X	X	X	LOAD C FROM BORROW
	C: = 0	X	X	1	1	0	X	X	X	X	X	X	X	X	LOAD 0 INTO C
	C: = 1	X	X	1	1	1	X	X	X	X	X	X	X	X	LOAD 1 INTO C
RAM	READ	0	0	X	X	X	X	X	X	X	X	X	X	X	READ FROM RAM
	RAM: = CM	0	1	X	X	X	X	X	X	X	X	X	X	X	LOAD RAM FROM CM
	RAM: = C	1	0	X	X	X	X	X	X	X	X	X	X	X	LOAD RAM FROM C
	RAM: = SM	1	1	X	X	X	X	X	X	X	X	X	X	X	LOAD RAM FROM SUM

Fig. 2

composte diverse.

Il simbolo di := ha lo stesso significato di assegnazione che gli viene attribuito in Pascal. La right-part dell'assegnamento può essere costituita da un registro, da un valore costante, dal contenuto della RAM o dal simbolo di una direzione (E, W, N, S).

Istruzioni come NS:=N o come EW:=W implementano la comunicazione con i p.e. vicini. Ad esempio, l'istruzione NS:=N provoca una traslazione dei registri NS verso Sud. Infatti ogni registro NS assumerà il valore del registro NS del p.e. immediatamente a Nord. Similmente EW:=W provoca uno shift di dati verso EST. A causa della loro rappresentazione visiva, queste istruzioni sono spesso chiamate operazioni orizzontali.

In contrapposizione a queste vi sono operazioni verticali: tali sono gli accessi in memoria. Notiamo subito che non tutti i registri di un p.e. possono scrivere in memoria. Tale possibilità è concessa solo al registro C ed a quello di comunicazione CM. È però possibile, grazie alla particolare implementazione hardware del chip, miscelare le istruzioni primitive per ottenere pseudo-istruzioni del tipo:

C:=NS RAM:=C.

L'effetto è quello che si otterrebbe con

RAM:=NS.

Finora non abbiamo fatto nessun accenno all'indirizzamento della memoria: ciò avviene anteponendo alla riga istruzione l'indirizzo di memoria su cui operare:

1: RAM:=C

A causa della particolare conformazione architetturale, si parla spesso di bit-planes e di immagini. Un bit-plane non è altro che un array di bit di memoria (nel nostro caso di dimensione 12x12) giacenti tutti allo stesso indirizzo. Un'immagine è un insieme di bit-planes giacenti su indirizzi contigui.

Per concludere la nostra rapida panoramica sull'assembler GAPP, esaminiamo il set di funzioni aritmetico-logiche messe a disposizione dal GAPP. Come si nota dalle figure 3 e 4 le funzioni aritmetiche possono essere viste co-

ARITHMETIC OPERATIONS Adder/Subtracter Operations					
INPUT			OUTPUT		
NS	EW	C	SM	CY	BW
0	0	0	0	0	0
0	1	0	1	0	1
1	0	0	1	0	0
1	1	0	0	1	0
0	0	1	1	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	1	1	1	1

Fig. 3

LOGIC OPERATIONS		
Logical Operation	Description	Conditions
INV	$SM = \overline{NS}$ $SM = \overline{EW}$ $SM = \overline{C}$	EW = 0, C = 1 NS = 0, C = 1 NS = 0, EW = 1
AND	$CY = NS \bullet EW$ $CY = EW \bullet C$ $CY = \overline{NS} \bullet EW$ $BW = \overline{NS} \bullet EW$	C = 0 NS = 0 C = 0 C = 0
OR	$CY = NS + EW$ $BW = \overline{NS} + EW$ $BW = EW + C$	C = 1 C = 1 NS = 0
XOR	$SM = NS \oplus C$ $SM = NS \oplus EW$ $SM = EW \oplus C$	EW = 0 C = 0 C = 1
XNOR	$SM = \overline{NS \oplus EW}$	C = 1

Fig. 4

me particolari funzioni logiche, sempre che siano verificate opportune condizioni iniziali. Abbiamo così a disposizione un set di istruzioni più che sufficiente anche per applicazioni di una certa complessità.

Vediamo, come rapido esempio, l'algoritmo per sommare due numeri binari. Supponiamo che gli addendi siano già in memoria GAPP, che abbiano la stessa dimensione in bits e di conoscerne gli indirizzi di memorizzazione (da 0 a 7 per il primo addendo, da 8 a 15 per il secondo e il risultato memorizzato da 16 a 24). Allora l'algoritmo è il seguente:

```
 0: C:=0 NS:=RAM; /* Clear carry and load 1st bit */
 8: EW:=RAM; /* Load 1st bit of second image */
16: C:=CY RAM:=SM; /* Store sum and save carry */
 1: NS:=RAM; /* Load 2nd bit of first image */
 9: EW:=RAM; /* Load 2nd bit of second image */
17: C:=CY RAM:=SM; /* Store sum and save carry */
 2: NS:=RAM; */ ... */
10: EW:=RAM; /* and so on */
18: C:=CY RAM:=SM; /* ... */
 3: NS:=RAM;
11: EW:=RAM;
19: C:=CY RAM:=SM;
 4: NS:=RAM;
12: EW:=RAM;
```

```
20: C:=CY RAM:=SM;
 5: NS:=RAM;
13: EW:=RAM;
21: C:=CY RAM:=SM;
 6: NS:=RAM;
14: EW:=RAM;
22: C:=CY RAM:=SM;
 7: NS:=RAM;
15: EW:=RAM;
23: C:=CY RAM:=SM;
24: RAM:=C; /* Finally store carry */
```

È ovvio che è impensabile esprimere tutti i programmi per il GAPP in questa forma. A tale scopo la NCR ha sviluppato un linguaggio, denominato GAL (GAPP Algorithmic Language), che aggiunge strutture di controllo C-like all'assembler del GAPP.

Grazie a questo linguaggio di alto livello, abbiamo a disposizione una più sofisticata gestione della memoria del GAPP e la possibilità di una programmazione resa più facile dalla presenza di strutture di controllo.

In particolare, viene definito il tipo di dati image. Una variabile di questo tipo risiede nella memoria GAPP, ed ha una dimensione in bit specificata all'atto della dichiarazione. Il compilatore si occupa di allocare la variabile in memoria GAPP in modo trasparente al programmatore, e di gestire tutti i riferimenti simbolici alla stessa. Esiste inoltre la possibilità di specificare espressamente l'indirizzo di partenza della variabile, così come quella di indirizzare in maniera assoluta la memoria del GAPP.

Dal punto di vista delle strutture di controllo messe a disposizione del programmatore, non crediamo sia necessario dilungarci sulla loro descrizione, in quanto esse sono in tutto simili a quelle presenti nel linguaggio C. Dobbiamo spendere due parole, invece, per quanto riguarda le espressioni che regolano l'esecuzione delle strutture di controllo.

Se in tali espressioni, ad esempio nella condizione di ingresso di un while, compaiono solo variabili intere, messe a disposizione del programmatore per implementare cicli e test, il loop viene espanso alla sequenza effettiva di esecuzione. Praticamente, dalla compilazione di

```
int i=1;
while(i++<7)
{
   A:i ram:=cm;
}
```

si ottiene

```
A:1 ram:=cm;
A:2 ram:=cm;
A:3 ram:=cm;
A:4 ram:=cm;
A:5 ram:=cm;
A:6 ram:=cm;
```

Come ulteriore caratteristica, valgono tutti gli operatori logici ed aritmetici del C, limitatamente alle variabili intere, ed esiste la possibilità di dichiarare funzioni che possono essere utilizzate allo stesso modo che nel C. In tal modo possiamo creare una libreria di funzioni, che possono operare su variabili di tipo image passate come argomento, e che possiamo richiamare dal programma principale.

Ricordiamo inoltre che le regole di visibilità delle variabili sono le stesse di quelle del linguaggio C.

A titolo di esempio riscriviamo il programma che somma due immagini in GAL.

```
main( )
{
image A:8, B:8, C:9; /* dichiaro le immagini A B C di */
                     /* dimensione 8, 8 e 9 bits */
somma (A,B,C);
}
somma(X,Y,R);
image X,Y,R;
{
   int i;
   C:=0;
   for(i=0;size(X);i++)
      {
      X:i NS:=RAM;
      Y:i EW:=RAM;
      R:i RAM:=SM C:=CY;
      }
   R:i RAM:=C;
}
```

La funzione size(X) ritorna la dimensione in bit dell'immagine X. Non sempre la sequenza di esecuzione di un programma sistolico può essere predeterminata così come per la somma. Può essere necessario valutare dei risultati intermedi di elaborazione. Per questo scopo il GAPP fornisce un segnale, denominato Global Output (GO), che rappresenta un OR logico di tutti i registri NS dell'array di processori.

Il GAL permette la valutazione run-time di questo valore tramite le due pseudo-funzioni gohigh e golow. Esse assumono valore di verità TRUE se il GO è a livello logico 1 o 0 rispettivamente.

A questo punto manca solo la possibilità di poter agganciare programmi sistolici con programmi con linguaggi ad alto livello come il C o il Pascal.

L'interfaccia software con i sistolici

L'idea di poter utilizzare algoritmi sistolici nella elaborazione delle immagini ci ha spinto ad approfondire alcuni aspetti hw/sw della scheda GAPSYS.

Innanzitutto, come caricare dati nella memoria sistolica? Un algoritmo che risolve il problema può essere il seguente:

Cycle #	Corner Turn Operation	GAPP Array Operation	Comment
1	EW := W	CM:= CMS*	Shift one row (i.e. 12 entries) of 8-bit
2	EW := W	or	data into the Corner Turn Line
3	EW := W	CM:= 0*	Buffer.
4	EW := W		
5	EW := W		
6	EW := W		
7	EW := W		
8	EW := W		
9	EW := W		
10	EW := W		
11	EW := W		
12	EW := W		
13	EW to CM	CM := RAM(0)	Prepare to shift all the least significant bits of the row to array.
14	CM := CMS	CM := CMS	Shift LSB into GAPP Array.
15	CM to EW	RAM(0) := CM	Wair for GAPP Array to store LSB
16	EW to CM	CM := RAM(1)	in RAM.
17	CM := CMS	CM := CMS	Shift 2nd LSB into GAPP Array
18	CM to EW	RAM(1) := CM	Wait for GAPP Array to store 2nd
19	EW to CM	CM := RAM(2)	LSB in RAM.
20	CM := CMS	CM := CMS	Shift 3rd LSB into GAPP Array
21	CM to EW	RAM(2) := CM	
22	EW to CM	CM := RAM(3)	
23	CM := CMS	CM := CMS	
24	CM to EW	RAM(3) := CM	
25	EW to CM	CM := RAM(4)	.
26	CM := CMS	CM := CMS	.
27	CM to EW	RAM(4) := CM	.
28	EW to CM	CM := RAM(5)	
29	CM := CMS	CM := CMS	
30	CM to EW	RAM(5) := CM	
31	EW to CM	CM := RAM(5)	
32	CM := CMS	CM := CMS	Shift 2nd MSB into GAPP Array.
33	CM to EW	RAM(6) := CM	Wait for GAPP Array to store 2nd
34	EW to CM	CM := RAM(7)	MSB in RAM.
35	CM := CMS	CM := CMS	Shift MSB into GAPP Array.
36	—	RAM(7) := CM	Store MSB.

* There are no restrictions on any other register operations in the GAPP Array for these cycles.

Come si può notare, il tutto funziona grazie al CTLB. Tale circuito ci permette la necessaria conversione di formato dei dati ed il cambio di direzione di shift da Est a Nord. Lo stesso discorso deve essere applicato al problema di estrarre dati dalla memoria del GAPP. L'algoritmo utilizzato è, praticamente, il complementare del precedente:

Cycle #	Corner Turn Operation	GAPP Array Operation	Comment
1	—	CM := RAM(0)	
2	CM := CMS	CM := CMS	Shift LSB into CTLB from the GAPP Array.
3	CM to EW	RAM(0) := CM	Wair for GAPP Array to get 2nd
4	EW to CM	CM := RAM(1)	LSB from RAM.
5	CM := CMS	CM := CMS	Shift 2nd LSB into CTLB from GAPP Array
6	CM to EW	RAM(1) := CM	Wait for GAPP Array to get 3rd LSB
7	EW to CM	CM := RAM(2)	from RAM.
8	CM := CMS	CM := CMS	Shift 3rd LSB into CTLB from GAPP Array
9	CM to EW	RAM(2) := CM	
10	EW to CM	CM := RAM(3)	
11	CM := CMS	CM := CMS	
12	CM to EW	RAM(3) := CM	
13	EW to CM	CM := RAM(4)	
14	CM := CMS	CM := CMS	
15	CM to EW	RAM(4) := CM	
16	EW to CM	CM := RAM(5)	
17	CM := CMS	CM := CMS	
18	CM to EW	RAM(5) := CM	
19	EW to CM	CM := RAM(6)	
20	CM := CMS	CM := CMS	Shift 2nd MSB into CTLB from GAPP Array.
21	CM to EW	RAM(6) := CM	Wait for GAPP Array to to get MSB from RAM
22	EW to CM	CM := RAM(7)	
23	CM := CMS	CM := CMS	Shift MSB into CTLB from GAPP Array.
24	EW := W	CM:= CMS*	Shift one row (i.e. 12 entries) of 8-bit
25	EW := W	or	data into the Corner Turn Line
26	EW := W	CM:= 0*	Buffer into the Output Data
27	EW := W		Register.
28	EW := W		
29	EW := W		
30	EW := W		
31	EW := W		
32	EW := W		
33	EW := W		
34	EW := W		
35	EW := W		

* There are no restrictions on any other register operations in the GAPP Array for these cycles.

Non analizziamo nel dettaglio i due algoritmi, ma è interessante fornire alcune cifre: per il download di una matrice di 12x12 bytes occorrono circa 420 istruzioni GAPP; con un clock a 10 MHz impiegheremmo qualche centinaio di microsecondi, contro un tempo che ora è di qualche centesimo di secondo.

Questi due algoritmi sono stati implementati prima in C e poi in assembler e fanno parte di una libreria di routines che si possono agganciare con alcuni linguaggi di alto livello in ambiente MS-DOS come il C ed il Pascal.

Upload

Funzione: Trasferisce dati dalla RAM del GAPP a un array utilizzabile dal programma. Va specificato l'array di destinazione (in generale un array di 12x12 char), l'indirizzo del GAPP dal quale si vuole leggere l'immagine (0...127) e la dimensione (quanti bit sono dedicati a ogni elemento dell'array) dell'immagine che si vuole leggere (1..8). Il bit più significativo viene preso dal piano più in basso della RAM del GAPP.

Sintassi: *upload(arrayname,GAPP__RAM__addres,size)*

Metodo: In pratica sono state esplicitate le istruzioni descritte e convertite negli appositi OUT su porte di I/O del PC, dopo essere state ottimizzate dal punto di vista della velocità. L'esecuzione di questa procedura per una immagine di 8 bit richiede circa 46000 colpi di clock del PC.

Download

Funzione: Effettua l'operazione inversa di upload. Va spe-

cificato l'array dal quale leggere i dati (di tipo image), l'indirizzo del GAPP al quale si vuole scrivere l'immagine (da 1 a 127) e la dimensione (da 1 a 8 bit) dell'immagine che si vuole scrivere. In pratica è la complementare dell'upload. Il bit più significativo di ogni elemento dell'array viene messo nel piano più in basso della RAM del GAPP.

Sintassi: *dowload(arrayname,GAPP__RAM__address,size)*

Metodo: Anche qui sono state esplicitate le istruzioni e ottimizzate per essere trasformate in una serie di OUT sulle varie porte che svolgano questa funzione. Il numero di colpi di clock necessari per l'esecuzione è circa uguale a quello della procedura di upload.

Una ulteriore routine che abbiamo implementato ci permette di caricare dati nel GAPP mentre ne scarichiamo altri. Grazie a ciò risparmiamo circa il 25% del tempo macchina necessario ad eseguire una operazione di download ed una di upload.

Chload

Funzione: Si passano come parametri a questa procedura due nomi di array (tipo image), un indirizzo e una dimensione. Questa procedura prende i dati contenuti nel primo array specificato e li scrive nel GAPP all'indirizzo fornito, mentre contemporaneamente l'immagine che era memorizzata nel GAPP a tale indirizzo viene copiata nel secondo array specificato. In pratica esegue un upload e un download contemporaneamente.

Sintassi: *chload(newimage, oldimage, GAPP__RAM__address, size)*

Metodo: Si è sfruttato il fatto che le due procedure di upload e download contengono una parte di codice comune che può essere condivisa. Si tratta dello shift verso nord dei dati della linea CM nelle due matrici di processori (GAPP e CTLB); la stessa operazione, mentre consente l'ingresso di dati da Sud ne effettua anche l'espulsione verso Nord. In questo modo, rispetto a 2 operazioni separate di upload e download, si ottiene un risparmio di tempo. Questa procedura richiede circa 70000 cicli di clock del PC (upload e download separate ne richiederebbero 92000).

Ora bisogna risolvere il problema di poter eseguire programmi sistolici pilotando l'esecuzione da un linguaggio di alto livello.

A tal proposito abbiamo sfruttato un'opzione del compilatore GAL. Infatti possiamo ottenere come risultato della compilazione un file ASCII che contiene solo istruzioni scritte in assembler del GAPP. Esse sono il risultato dello "srotolamento" delle strutture di controllo e dei loops nei quali non compaiono tests che coinvolgono il global output.

Partendo da questi files è molto più facile ricavare i dati che ci permettono di eseguire un programma. In pratica è stato costruito un semplice assemblatore che legge il file, lo assembla in un particolare formato e ritorna al programma host l'indirizzo di partenza del programma così "compilato".

Per contraddistinguere il risultato della compilazione, abbiamo creato il tipo di dati G__PROG. Altro non è che una stringa di caratteri, nella quale i primi due bytes contengono informazioni sulla lunghezza del programma. Una volta "assemblato" il programma sistolico, possiamo eseguirlo abbastanza semplicemente. Ciò viene realizzato ponendo gli opportuni valori sulle porte di I/O del PC e fornendo via software il clock per l'esecuzione dell'istruzione. I risultati di questo lavoro sono due functions C che ora descriviamo.

Compile

Funzione: Compila i programmi scritti in GAL preprocessati dal compilatore "CG" (con l'opzione -a) fornito dalla NCR nel package GAPSYS. A questa procedura viene passato il nome di un file contenente il programma da compilare e viene restituito un puntatore a una lista di byte che rappresenta il formato eseguibile del programma. È incluso un analizzatore sintattico che segnala eventuali errori: in questo caso viene stampato un messaggio di errore ma la compilazione continua. Non sempre la compilazione viene effettuata perchè può verificarsi l'eventualità che la memoria disponibile sia stata esaurita; in questo caso viene stampato un WARNING e la procedura ritorna il valore NULL che può venire testato dal programma chiamante. Il numero massimo di righe compilabili è di circa 18000; tale numero viene ridotto proporzionalmente all'area dati occupata dal programma chiamante. Il programma compilato può essere poi eseguito dalla procedura exec__prg.

Sintassi: *prg=compile("nomefile")*

con prg definito di tipo G__PROG.

Metodo: Il compilatore "CG" con l'opzione -a trasforma tutte le strutture di controllo dei programmi scritti in GAL rendendole esplicite. In pratica dopo la compilazione si ha un programma lineare (cioè privo di loop, if ecc.) dove tutti i loop sono stati "srotolati". Da un programma di tipo lineare la procedura di compile genera una stringa di bytes formata dai valori che devono essere scritti sulle porte P0, P1, P2 affinché il programma sia eseguito dai GAPP. I primi due byte della stringa contengono la lunghezza della stringa stessa. Dato che per ogni istruzione sono richiesti 3 valori per le porte di OUT, il numero di istruzioni contenute nelle stringa è dato da (*prg*[0] + 256 * *prg* [1] — 2)/3 dove prg[0] e prg[1] sono iprimi 2 byte della stringa e la costante 2 che viene sottratta serve per togliere dal calcolo delle istruzioni proprio questi 2 byte. In prg[i] c'è il valore che deve essere scritto sulla porta:

$$P0 \; se \; (i - 2) \; mod \; 3 = 0$$

$$P1 \; se \; (i - 2) \; mod \; 3 = 1$$

$$P2 \; se \; (i - 2) \; mod \; 3 = 2$$

Exec_PRG

Funzione: A questa procedura viene passato il puntatore fornito dalla compile e il programma contenuto in tale stringa viene eseguito dai GAPP.

Sintassi: *exec_prg(prg)*

con prg definito di tipo G_PROG.

Metodo: La stringa puntata da prg viene scandita e i valori letti vengono scritti sulle porte P0, P1, P2. Dopo che ogni terna di valori è stata scritta viene attivata la porta P7 che è quella che genera il clock per la matrice di GAPP. Solo a questo punto l'istruzione viene effettivamente eseguita. Per l'esecuzione di ogni istruzione (una riga di programma) sono necessari 160 colpi di clock.

Come ulteriore utility abbiamo aggiunto una routine che ci permette di caricare i programmi già in forma eseguibile da file, senza eseguire compilazioni non necessarie.

Take_GPROG

Funzione: Questa procedura si può definire un'utility da affiancare alla compile. In effetti la take_GPROG si comporta come una compile, con la differenza che il risultato della compilazione viene anche salvato su disco. Quando viene richiamata, questa routine controlla su disco se esiste già la versione compilata del programma specificato come argomento. In questo caso viene caricato direttamente. Se tale versione non esiste, il programma viene compilato e la sua versione compilata viene salvata su disco con il suffisso .XG.

Sintassi *prg=take_GPROG("nomefile")*

con prg definito di tipo G_PROG.

A questo punto siamo quasi pronti per ottenere risultati di elaborazioni sistoliche in forma tangibile. Per dare una veste più bella a questi risultati, abbiamo provveduto ad interfacciare il nostro driver per sistolici con una scheda grafica a colori ad alta risoluzione.

In pratica abbiamo fatto in modo di poter leggere dati direttamente dalla memoria grafica e scriverli nella memoria del GAPP in una sola passata, e viceversa. La scheda grafica presa in considerazione è la ITEX color Graphic.

XUPLOAD

Funzione: Questa procedura prende l'immagine di 8 bit memorizzata nel GAPP all'indirizzo specificato come terzo parametro e la visualizza sul monitor grafico alla posizione X. Y (primo e secondo parametro).

Sintassi: *xupload(x, y, GAPP_RAM_address)*

Metodo: In pratica viene effettuato un upload con la differenza che i dati non vengono scritti in un array 12*12 ma in una finestrella 12*12 sulla memoria grafica della scheda ITEX. Questo consente una maggiore velocità rispetto a un upload seguito da un'operazione di scrittura dei dati dell'array sul video.

XDOWNLOAD

Funzione: È la procedura complementare di xupload. Viene letta una finestra di dimensioni 12*12 alle coordinate X, Y e copiata nella RAM del GAPP all'indirizzo specificato come terzo parametro.

Sintassi: *xdownload(x, y, GAPP_RAM_address)*

Metodo: Valgono le stesse considerazioni fatte per xupload.

Una volta in possesso di tutti gli strumenti necessari, abbiamo applicato l'elaborazione sistolica alle immagini. Ecco un breve sommario dei risultati ottenuti nel filtraggio di immagini:

Programma	tempo in GAL	tempo in C
banda passante	50 sec.	53 sec.
escludi banda	50 sec.	53 sec.
contrasto	58 sec.	61 sec.
inverse image	50 sec.	52 sec.
gradazione in 4 colori	86 sec.	73 sec.
gradazione in 9 colori	134 sec.	73 sec.
gradazione in 13 colori	161 sec.	73 sec.
filtro di Sobel	77 sec.	360 sec.
rilievo	52 sec.	90 sec.

3. PRESTAZIONI, LIMITI E POTENZIALITÀ

Durante questo lavoro somo emersi alcuni dati molto importanti per gli sviluppi futuri del progetto.

Il principale collo di bottiglia che è stato rilevato riguarda il passaggio di dati da/alla scheda. Ad esempio per elaborare un'immagine di 504*504 pixel sono necessarie 504*504/(12*12) = 1764 operazioni di upload e altrettante di download. Tenuto conto che le procedure di upload e download richiedono poco più di 10 millisecondi per la loro esecuzione (tempo questo che riteniamo non sia possibile migliorare ulteriormente a causa del tipo di hardware di questa scheda), risulta che per l'elaborazione di un'immagine occorrono circa 35 secondi di solo trasferimento di dati tra la memoria video e i GAPP. A questo punto è chiaro che per elaborazioni molto semplici (pochi calcoli) diventa inaccettabile pagare questo tributo di 35 secondi, mentre per elaborazioni complesse (5000 e più istruzioni GAPP) il peso di questa costante diventa meno rilevante.

Anche la dimensione di 12*12 GAPP è una severa limitazione. Infatti qualsiasi elaborazione da effettuare su grandi matrici di dati richiede che il programma venga eseguito

più volte, ogni volta su una finestra di 12*12.

Tornando all'esempio dell'immagine da 504*504 pixel, un programma di 1000 istruzioni GAPP deve essere ripetuto per 1764 volte, per un totale di 1764000 istruzioni. Utilizzando array di GAPP di dimensioni più ampie si potrebbe ottenere un importante miglioramento. Ad esempio con una matrice di 48*48 GAPP le prestazioni migliorerebbero di un fattore 16.

Il terzo collo di bottiglia che si è riscontrato riguarda i problemi di clock. Dopo molto lavoro si è riusciti a portare il tempo di esecuzione di una singola istruzione GAPP a circa 160 colpi di clock del PC, ch equivale a circa 33μsec su macchine operanti a 4.77 Mhz. Questo limite non è ulteriormente migliorabile a causa della struttura hardware della scheda fornita dalla NCR, che non è dotata di un clock proprio ma deve ricevere il clock dalle porte di I/O della CPU 8086.

Inoltre la scheda non è dotata di un controller (memoria di microprogramma), quindi ogni istruzione, per essere eseguita deve essere letta dal PC e scritta su tre porte di OUTPUT. È come se avessimo a che fare con una scheda il cui clock è di 30 Khz.

Per superare questo limite l'unica possibilità è quella di realizzare una scheda hardware diversa, che sia dotata di un clock proprio e una RAM dedicata dove memorizzare i programmi per i GAPP. Così facendo, se il clock fosse a 10Mhz, il tempo di esecuzione di una istruzione sarebbe 100 nanosecondi: un valore 330 volte più piccolo del limite attuale!

Questi risultati sono molto importanti se si considera il fatto che, pur con le limitazioni attuali e con la dimensione ridicola di soli 12*12 GAPP, alcune routine grafiche di filtraggio (Sobel, gradiente ecc. ...) hanno dato tempi di esecuzione leggermente inferiori alle stesse routine scritte con i metodi tradizionali.

Con un certo ottimismo potremmo dire che con una scheda dotata di clock proprio ed una matrice di 48*48 GAPP, si potrebbero raggiungere tempi circa 5000 (!) volte inferiori ai tempi di elaborazione ottenuti con macchine della classe PC-IBM e con linguaggi tradizionali.

4. RINGRAZIAMENTI

Desideriamo ringraziare il Prof. Gianni Degli Antoni per gli stimoli e le idee di fondo da cui è partito. Inoltre vogliamo ringraziare Liana Benassi e Roberto Grande per i preziosi consigli durante la stesura dell'articolo.

RAY TRACING E ARCHITETTURE SISTOLICHE

Stefano Zanantoni

Dipartimento di Scienze dell'Informazione

Università di Milano

RIASSUNTO

Si presenta un algoritmo sistolico per la esecuzione del passo di calcolo di intersezione raggio/oggetto nel metodo di ray tracing per la simulazione visiva. L'algoritmo si basa sulla ricerca degli zeri nel polinomio risolutore mediante il metodo derivato dal cambiamento dei segni (metodo di Cartesio). L'algoritmo è stato implementato su una scheda GAPP-NCR e si presentano alcuni risultati quantitativi.

ABSTRACT

In the paper a systolic algorithm is presented for the computation of the ray/object intersection step in ray tracing algorithm for visual simulation. The algorithm is based on root finding in the polynomial, by means of sign rule (Cartesio Method). The algorithm has been implemented on a GAPP-NCR systolic board, and some quantitative results are discussed.

1. PROCESSO DI VISIONE E RAY TRACING

La sintesi di immagini consiste nella creazione di immagini a partire da modelli geometrici curandone il grado di realismo. Per far questo esistono differenti algoritmi ma il metodo più accurato che permette di ottenere un elevato grado di realismo è quello del ray tracing, che richiede però tempi di elaborazione molto elevati.

Nel processo di visione i raggi di luce molteplicemente riflessi e rifratti dagli oggetti costituenti la scena giungono all'osservatore, determinando l'immagine che questi percepisce. Se si volesse simulare questo procedimento occorrerebbe generare infiniti raggi provenienti dalle sorgenti di luce, seguirli lungo le diverse traiettorie nello spazio, e, per i raggi che incontrano degli oggetti, determnare, in base alle leggi dell'ottica geometrica e alle proprietà fisiche dei singoli oggetti, il tipo di raggi generati dall'interazione raggio/oggetto (raggi riflessi e rifratti) e seguire questi ultimi come se fossero i raggi originali finché o si perdono "all'infinito", o sono totalmente assorbiti dagli elementi che via via incontrano.

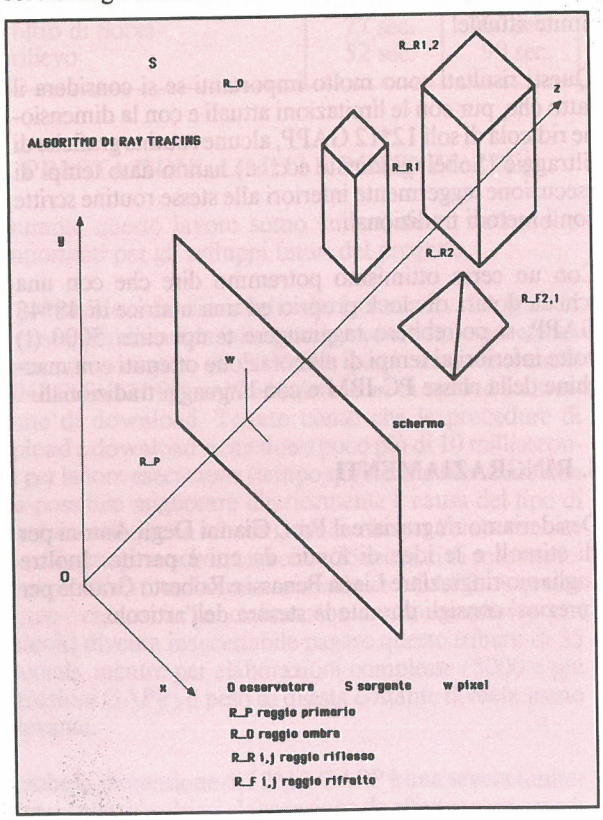

Fig. 1

Nella pratica questo processo viene notevolmente semplificato partendo dalla considerazione che i soli raggi determinanti l'immagine nell'osservatore sono quelli che ad esso giungono. Inoltre, poiché l'immagine generata ha come destinazione lo schermo dell'elaboratore, è possibile limitarsi a considerare solo i raggi che, passando per un punto dello schermo, il pixel, arrivano all'occhio di chi guarda. Il metodo di ray tracing è realizzato tracciando innanzitutto un raggio dall'osservatore ad ogni pixel e determinando, se esiste, l'intersezione con uno o più oggetti costituenti la scena. Questi raggi non sono reali ma geometrici e prendono il nome di raggi primari. Per ogni punto di intersezione individuato occorre tracciare un raggio riflesso e/o uno rifratto, nell'ipotesi semplificativa che vi siano al più due nuovi raggi, e considerare questi ultimi come se fossero nuovi raggi primari. La decisione se un punto deve apparire illuminato oppure no è presa tracciando da quel punto verso le sorgenti di luce un nuovo raggio, detto raggio ombra, e individuando se si frappongono oggetti opachi lungo la direzione presa. Il procedimento esposto è ricorsivo e si arresta o per "morte" del raggio o perchè si è soddisfatti del numero di raggi non primari generati. Per ogni singolo pixel viene generato un albero binario, che rappresenta la storia dei raggi dopo l'intersezione. Il colore e l'intensità del pixel sono determinati in base ai contributi dei singoli nodi dell'albero.

L'elevata complessità computazionale deriva proprio dall'altissimo numero di raggi da generare; il calcolo delle intersezioni occupa dal 65% all'80% del tempo di elaborazione totale. Analizzando più attentamente il metodo del ray tracing si nota che tra tutte le intersezioni possibili tra un raggio ed un oggetto, solo la prima è necessaria, cioè quella che rappresenta il primo incontro raggio/oggetto (supposto il sistema di riferimento fissato nell'osservatore con l'asse z positivo verso la scena). Ciò corrisponde a calcolare la prima radice positiva dell'equazione polinomiale, che si ottiene sostituendo le equazioni parametriche di un raggio nella equazione che rappresenta la superficie dell'oggetto in esame, e utilizzando il valore del parametro ottenuto nella equazione del raggio.

2. L'ALGORITMO MODIFICATO DI COLLINS-AKRITAS

Il valore dello zero può essere ottenuto con vari algoritmi di tipo numerico, tra i quali quello derivato dal metodo di Collins e Akritas. Questo metodo è basato sulla Regola dei Segni di Cartesio e sul teorema di Vincent attraverso i quali si determinano delle trasformazioni, che, applicate al polinomio di grado n da risolvere, permettono, mediante bisezioni successive, di individuare, se esiste, l'intervallo nel quale di trova la più piccola radice positiva.

Il metodo Collins ed Akritas 'puro' [Coll 23] presenta dei problemi di implementazione che lo rendono inadatto ai nostri scopi.

Questi problemi si possono brevemente riassumere in due punti:

— è un metodo ricorsivo

— ricerca tutte le radici nell'intervallo assegnato.

A fronte di questi svantaggi ci sono degli aspetti interessanti, come lo scarso utilizzo di operazioni floating point, la modularità, la possibilità di riutilizzare parti di codice già scritte per compiti apparentemente dissimili, che lo fanno emergere, tra quelli analizzati, come il migliore algoritmo da implementare su una macchina sistolica.

Per far questo, però, bisogna eliminare i difetti.

A ciò ha provveduto Tampieri [Tamp 86]. Una soluzione è stata quella di riscrivere il flusso di istruzioni che compongono l'algoritmo semplificandolo ulteriormente.

L'algoritmo risultato è in grado di ricercare, in modo iterativo, la minore radice positiva.

Per ridisegnare l'algoritmo si introduce una nuova funzione, chiamata "Choice" così definita:

Choice

input: A(t) polinomio in grado n con radici nell'intervallo (0,1).

output: R(t) polinomio di grado n trasformato e un flag b_i settato a 0 o ad 1.

{ b_i rappresenta l'(n —1) — esimo bit della radice cercata. Questo significa che i bit sono emessi da quello più significativo a quello meno significativo }

1) [**Metà sinistra**] si calcoli

$$A'(t) = 2^n * A(t/2)$$

{ gli zeri di A(t) nell'intervallo (0,1/2) sono trasformati in quelli di A' (t) in 0,1) }

2) [**Metà destra**] si calcoli

$$A''(t) = A'(t+1)$$

{ gli zeri di A(t) in (1/2,1) sono trasformati in quelli di A" (t) in (0,1) }

3) [**Scelta**] se

var $((1+t)^n * A'(1/(t+1))) > 0$ allora V1 = 1

{ se V1 è vero allora esiste almeno una radice nel sottointervallo sinistro: con "var" denotiamo il numero di variazioni di segno del polinomio nell'intervallo considerato }

se

$A(1/2) = 0$ allora V2 = 1

{ se V2 è vero allora esiste una radice tra i due sottointervalli }

se

var$((1+t)^n * A''(1/(t+1))) > 0$ allora V3 = 1

{ se V3 è vero allora esiste almeno una radice nel sottointervallo destro }

4) [**Conclusione**] il flag emesso b_i e il nuovo polinomio R(t) sono selezionati secondo questa tavola di verità:

V1	V2	V3	b_i	R(t)
1	0	0	0	A' (t)
0	1	0	1	A' (t)
0	0	1	1	A" (t)
0	0	0	0	A" (t)

{ se esiste una radice nel sottointervallo sinistro si ricerca la radice in questo sottointervallo studiando il polinomio A' (t) e si emette il bit b_i pari a 0
altrimenti se la radice giace a metà tra gli intervalli si studia ancora A' (t) ma il flag è settato ad 1
altrimenti se esiste una radice nel **sottointervallo destro** si ricerca la radice in questo intervallo studiando il polinomio A" (t) e si emette il bit b_i pari a 1.
Infine se non vi è **nessuna radice,** oppure essa è già stata **trovata,** si fa in modo che Choice la approssimi da destra studiando A" (t) e il bit a 0. }

Fine CHOICE

Dato un qualsiasi polinomio, prima di poter applicare la funzione Choice occorre trasformarlo, in accordo coi primi passi dell'algoritmo da cui è stato ricavato, in modo che tutte le sue radici siano nell'intervallo (0,1).

Il numero di iterazioni deve essere stabilito in base alla precisione desiderata. Se si vuole che l'ultimo intervallo abbia ampiezza massima

$$\epsilon = 1/(2^m)$$ sono necessari almeno:

$$m > -\log(\epsilon)$$

iterazioni o equivalentemente m sistemi che realizzino Choice collegati in pipeline.

Gli m flag emessi (b_{m-1}, ... , b_o) costituiscono le m cifre binarie, da quella più significativa a quella meno significativa, del numero d'ordine dell'intervallo che contiene la radice cercata.

Il risultato finale è un intero tra 0 e $2^m - 1$ e viene ricavato in questo modo:

$$rad = B_{m-1}*2^{m-1} + b_{n-2}*2^{m-2} + ... + b_1*2 + B_o$$

e quindi:

$$radice = rad/(2^m - 1)$$

questa radice ha un valore compreso tra (0,1). Per ottenere la radice del polinomio da cui siamo partiti occorre ricordarsi di moltiplicare il valore ottenuto per il root bound usato per la normalizzazione.

La scelta di implementare l'algoritmo su architetture sistoliche è dovuta al fatto che il criterio di individuazione dell'intervallo si basa sulla valutazione delle variazioni e delle permanenze dei segni dei polinomi via via ottenuti. In tal modo il numero di operazioni "floating point" è molto ridotto. L'architettura sistolica utilizzata è quella della scheda NCR_GAPP versione di 12x12 PE (o elementi di elaborazione) disposti ad array bidimensionale "mesh connected" (ogni PE può comunicare con i propri quattro vicini) e con la possibilità per i PE di frontiera di comunicare o ricevere dati a/da quelli della frontiera, secondo una tipologia toroidale.

3. STRUTTURA DEL PROGRAMMA SISTOLICO

Si è scelto di operare in termini di virgola fissa perchè l'aritmetica 'floating point' non può essere implementata efficientemente perchè le operazioni da effettuare dipendono dai numeri su cui si opera.

3.1 Struttura dati

Ogni numero è composto da 32 bit in rappresentazione a complemento a 2. Questi sono intesi come insieme di bit e non numeri quindi il loro significato può essere di interi o reali.

I polinomi

I polinomi sono disposti nell'array 12*12 in modo da massimizzare lo sfruttamento dello spazio a disposizione per eseguire il maggior numero di ricerche di radici contemporanemanete. Questo significa che potenzialmente è possibile trattare un egual numero di pixel dello schermo in modo parallelo.

Il coefficiente relativo alla massima potenza della variabile del primo polinomio è posto nella locazione 0,0 cioè nell'angolo nord-ovest della scheda, quelli rimanenti sono disposti a fianco del primo in ordine di potenza decrescente e il termine noto completa il polinomio.

Se lo spazio a disposizione sulla riga dell'array lo consente ne vengono aggiunti altri fino al completamento della stessa.

Le righe successive seguono lo schema della prima. Lo spazio totale occupato dai polinomi in termini di piani di bit è k*k*32. L'insieme di tali piani sarà indicato col termine di immagine.

In pratica significa che se l'array è di dimensione k*k e il grado dei polinomi in esame è n con $n + 1 = k$, si riescono a sistemare su una riga fino a:

q (numero polinomi su una riga) = (k) / (n + 1)

e sull'intero array:

Q (numero polinomi sull'array) = k*q

dove il simbolo "/" rappresenta la divisione intera.

I flag

I flag sono piani di bit di dimensione k*k*1 e sono utilizzati come maschere per abilitare e disabilitare i PE. Altri raccolgono il risultato del computo delle variazioni e dei bit emessi a fine ciclo.

Il risultato

Il risultato è formato da k*k*16 elementi ed è disponibile solo alla fine delle iterazioni. È composto dal raccoglimento dei piani di bit emessi come risultato alla fine di ogni ciclo.

Per trovare l'effettiva radice richiesta occorre dividere il risultato per la potenza di due elevato al numero di cicli effettuati e moltiplicarlo per il root bound selezionato. Inoltre se i coefficienti non sono interi occorre shiftare verso sinistra (cioè dividere) di tanti passi quanti bit formano la parte frazionaria massima consentita.

Assieme alle strutture dati elencate, proprie dei programmi GAL, ne esistono altre, relative ai programmi C, utilizzati dall'host per gestire gli algoritmi.

Come memoria intermedia si sono scelti gli array di k*k unsigned long (la cui dimensione è 32 bits) oppure di dimensione inferiore (unsigned char pari a 8 bit), utilizzati per mantenere temporaneamente i risultati parziali ottenuti in termini di polinomi trasformati, in modo da liberare parte dei 128 bit disponibili come static RAM nel sistolico.

3.2 Risoluzione sistolica del problema delle radici

In questa sezione parleremo dell'implementazione dell'algoritmo per la ricerca della prima radice reale positiva di polinomi di grado n tra uno e 'più infinito'.

I punti cruciali del metodo utilizzato sono tre:

— il calcolo di A(t+1) dato A(t),

— quello delle variazioni dei polinomi trovati e

— il calcolo della normalizzazione.

Esistono altri punti oltre a quelli che saranno approfonditi, ma poiché la loro implementazione non si discosta molto da quelli presentati ed è facilmente intuibile, ne è omessa la discussione.

Nella descrizione degli algoritmi si è preferito evidenziarne i passi logici facendo ricorso ad una scrittura facilmente intuitiva.

Le notazioni usate hanno i seguenti significati:

size (A) : è un numero (in notazione decimale o ottale) che rappresenta la dimensione di un blocco di piani, cioè il numero dei piani che lo compongono. Se A è una immagine size(A) è pari a 32 (decimale) se invece rappresenta un flag il suo valore è 1 (decimale).

& : rappresenta AND logico tra i piani

! : rappresenta OR logico tra i piani

inv(A) : rappresenta il COMPLEMENTO ad uno di A

<—— : simbolo di ASSEGNAMENTO di uno o più piani ad un insieme di altri piani

= : operatore di EGUAGLIANZA tra variabili numeriche.

Il passo di SOSTITUZIONE

Il calcolo di A(t+1) eseguito direttamente, cioè sostituendo alla variabile t la nuova variabile t+1, elevando a potenza e raccogliendo le potenze di t, è senza dubbio irrealistico per dei processori ad un bit con poca memoria a disposizione.

Nella effettiva implementazione si utilizzano tre immagini (dimensione k*k*32) ed un flag A, B, C e F rispettivamente.

Nel blocco C sono sistemati i coefficienti dei polinomi nel modo che è stato già definito, A e B sono usati per contenere i risultati parziali e finali, rispettivamente, e il flag F permette il corretto caricamento dei coefficienti in B.

Per chiarire il funzionamento dell'algoritmo supponiamo di lavorare contemporaneamente su tutti i bit che compongono i coefficienti dei polinomi di grado n messi in C, con B e A inizialmente vuoti e F con 1 sulla colonna dei coefficienti relativi alle potenze di t di grado massimo e nei rimanenti punti posto a valore 0.

Lo schema dell'algoritmo è il seguente:

CALCOLO della SOSTITUZIONE t <—— T + 1 per polinomi di grado n

input : immagine C contenente i coefficienti dei polinomi, il flag F con gli uni in corrispondenza della massima potenza dei polinomi

output : immagine B con i risultati della sostituzione messi lungo le diagonali

FOR i = 0 TO grado — 1

1) **[load coef]** carica in C i coefficienti mediante l'operazione

B <—— ! (C & F)

2) [**add**] Calcola la somma di A e B e mette il risultato sia in A che in B

$$A = B <\!\!-\!\!- A + B$$

3) [**shift**] Per simulare il movimento dei dati lungo i canali si esegue lo shifting dei blocchi A e B e del flag F

$$B <\!\!-\!\!-$$ B shifta B verso SUD

$$A <\!\!-\!\!-$$ A shifta A verso EST

$$F <\!\!-\!\!-$$ F shifta F verso EST

FINE CALCOLO della SOSTITUZIONE.

Poiché il risultato ottenuto è disposto in diagonale e quindi in posizione non corretta rispetto a quella iniziale dei dati occorre utilizzare la seguente procedura per il riposizionamento.

RIPOSIZIONAMENTO COEFFICIENTI di polinomi di grado n

input : immagine B con i valori da riposizionare e il flag G con gli uni relativi ai termini noti

output : immagine B con i valori posizionati correttamente

FOR i = 0 TO grado — 1

1) [**inversione**] Si inverte il contenuto G

$$K <\!\!-\!\!-$$ inv(G)

2) [**separazione**] Si separano in due piani gli elementi che sono nelle posizioni corrette (in Y) da quelli da spostare (in Z)

$$Y <\!\!-\!\!-$$ B & G

$$Z <\!\!-\!\!-$$ B & K

3) [**shift**] Si spostano di un passo verso la posizione corrente tutti i coefficienti ancora fuori posto

$$Z <\!\!-\!\!-$$ Z shiftato verso NORD

4) [**ricostruzione**] Si ricostruisce l'immagine B nella quale ora un'altra colonna di coefficienti è posizionata correttamente

$$B <\!\!-\!\!-$$ Z : Y

5) [**mascheratura**] Si mascherano le posizioni che non si devono più modificare duplicando il flag che le controlla in direzione dei coefficienti relativi al termine di grado più elevato

$$G <\!\!-\!\!-$$ G : (G shiftato a OVEST)

FINE RIPOSIZIONAMENTO COEFFICIENTI

Con l'algoritmo descritto l'azione svolta è letteralmente quella di 'tirare verso l'alto' le colonne di tutti i coefficienti fuori posto facendo in modo da sistemare quelle che lo sono per una sola casella e avvicinando di un passo quelle che sono più distanti.

Poiché la distanza massima dei coefficienti fuori posto è grado —1 posizioni, con altrettanti 'strattoni' si ottiene la configurazione voluta.

Calcolo delle variazioni

Il modo più efficiente per eseguire il calcolo delle variazioni dei polinomi ottenuti è di concentrarsi nel piano relativo al segno degli elementi in esame ma, a questo livello, un numero positivo è indistinguibile dallo zero. Non essendo possibile indirizzare una unica celletta dell'array di processori ma solo dei piani k*k, l'idea utilizzata è stata di riprodurre in verticale, cioè su più piani, il numero che si legge sul piano dei segni considerandoli come cifre binarie.

Quello che si ottiene è una immagine di dimensione k*k (numero_coef) nella quale il piano dei segni è inteso come codifica binaria di numeri tra 0 e $2^{numero_coef\ -1}$.

Il confronto con dei valori di soglia serve a stabilire se esiste almeno una variazione oppure se non ne esiste alcuna.

Nel primo caso si confronta l'immagine con la soglia nulla e nel secondo con una soglia pari al massimo numero rappresentabile meno uno.

Con questo procedimento si ottengono due flag W e Y e quest'ultimo contiene uni ad indicare se la soglia **non** è stata superata. L'intersezione tra i flag identifica quali polinomi hanno possibili variazioni.

Il risultato ottenuto è affetto da errore perchè sequenze del tipo 10111 e 00000 sono ambigue. Infatti possono essere intese come una successione di —+———— e +++++ oppure —0———— e 0+0+0 data l'indistinguibilità dello 0 dal segno del numero positivo. Per eliminare questo problema occorre che sui coefficienti originali sia preventivamente eseguito un test di nullità, che il risultato sia preso come piano dei segni e si riapplichino i passi relativi ottenendo dei nuovi flags.

Le informazioni contenute nei flag sono ora sufficienti per stabilire con certezza se esistono o meno delle variazioni nei polinomi originari. Il risultato trovato mediante le operazioni booleane è immagazzinato in un nuovo piano per essere successivamente utilizzato nella determinazione della radice.

L'insieme dei passi, a livello logico, è descritto di seguito:

CALCOLO delle VARIAZIONI di polinomi di grado n

input : immagine B contenente i coefficienti dei polinomi sistemati nella maniera standard e il flag F con uni

corrispondenti ai coefficienti relativi ai termini di massimo grado

output: il flag VV contenente uni in corrispondenza dei coefficienti relativi ai termini di massimo grado se almeno una variazione è presente in quel polinomio

1) [**conversione**] Si utilizza il piano dei segni per convertire in numeri i segni dei polinomi

A <—— B: size (B) —1 indica il caricamento in A del piano di segni di B

FOR i = 1 TO (numero__coef —1)

1.1) T <—— A shifta A verso OVEST

1.2) B: size (B) —i —1 <—— t & B: size (B) —i —1

Ora negli ultimi numero__coeff piani di B vi è il risultato dei passi precedenti.

2) [**soglie**] si conferma ciò che si è ottenuto con i valori di soglia 0 e $2^{\text{numero_coef}-1}-2$

W <—— risultato di: maggiore della soglia 0? (vero = *1, falso* = 0)

Y <—— risultato di: minore della soglia $2^{\text{numero_coef}-1}-2$? (vero = 1, falso = 0)

3) [**intersezione**] Si esegue l'intersezione tra i flag per individuare quali coefficienti hanno almeno una variazione e si eliminano i risultati inutili mediante il flag F

VV <—— Y & W

J <—— VV & F

4) [**eliminazione**] Per eliminare gli eventuali casi non considerati si esegue un test sulla nullità di coefficienti originari in B e si tratta il risultato nel flag Z coi passi 2) e 3) ottenendo rispettivamente W e Y

5) [**composizione**] Il risultato finale si ottiene con le seguenti operazioni logiche

Z <—— W & Y & F

VV <—— J & Z

FINE CALCOLO delle VARIAZIONI

3.3. La normalizzazione

La realizzazione di questo passo presenta un problema analogo a quello che abbiamo affrontato nel riposizionamento dei coefficienti polinominali: la necessità di operare in modo differenziato sui dati a disposizione.

Nel caso in esame occorre moltiplicare per differenti potenze di due i coefficienti dei polinomi.

Supponiamo che A sia l'immagine di partenza e B quella che conterrà il risultato. Inoltre sia F il flag che abbiamo già utilizzato. L'algoritmo che esegue il passo di normalizzazione deve aggiungere un numero di zeri pari alla potenza di due del fattore di moltiplicazione k. Questi è determinato dal prodotto del grado del coefficiente considerato per il valore di normalizzazione scelto.

Al coefficiente relativo alla potenza di grado massimo vanno aggiunti grado*k zeri, ne occorrono (grado —1)*k per quello che lo segue in ordine decrescente e così via, lasciando inalterato il solo termine noto. Nella descrizione dell'algoritmo si deve far ricorso e dei pedici che indicheranno il piano relativo all'immagine che si sta usando.

NORMALIZZAZIONE POLINOMI di grado n

input : immagine A contenente i valori iniziali e il flag F con gli uni posti sui coefficienti relativi alla potenza maggiore dei polinomi. Si suppongono noti il grado dei polinomi (grado) e il valore di normalizzazione (k)

output : immagine B contenente i valori normalizzati

a) FOR h = 0 TO numero__coef —1

a.1) [**segno**] Copia il segno dei numeri in A in quelli di B. (La moltiplicazione non altera il segno)

$$B_{\text{size (B)}-1} \text{ <—— } A_{\text{size (A)}-1} \text{ e } h = 0$$

b) FOR i = 0 TO (size(A) —2 —j)

b.1) [**colonna scelta**] Isola la colonna da trattare sul piano i__esimo di A

$$W \text{ <—— } F \,\&\, A_I$$

b.2) [**posizionamento**] Posiziona W in B all'altezza i+j

$$B_{I+J} \text{ <—— } W : B_{I+J}$$

a.2) [**nuova posizione**] Sposta F nella direzione dei coefficienti di grado minore di un passo e calcola il nuovo j

F <—— F shifta F verso EST

h++ e j = k*grado*k

FINE NORMALIZZAZIONE

La normalizzazione eseguita con questo algoritmo poggia sulla ipotesi che l'insieme dei polinomi da trattare abbia radici contenute nello stesso intervallo (root bound identico). Benché a prima vista ciò possa far pensare ad una grave limitazione si deve tenere a mente che l'elaborazione sistolica richiede che le azioni sui dati siano comuni per raggiungere la massima efficienza. Il valore di root bound da applicare è quindi unico per ogni blocco di polinomi.

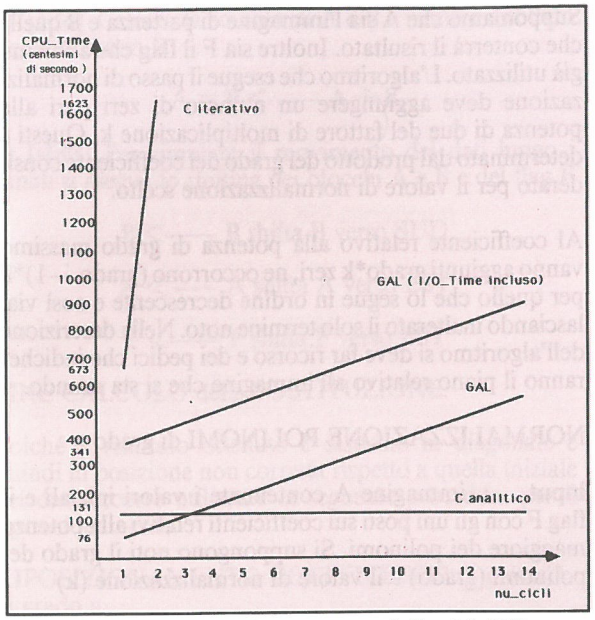

Fig. 2 Ipotesi 4.77 MHz programmi C - 30 KHz programma GAL.

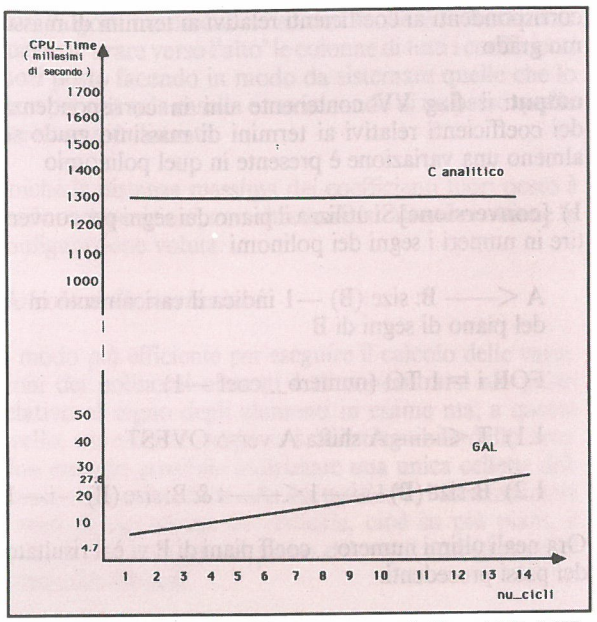

Fig. 3 Ipotesi: 4.77 MHz programmi C - 4.77 MHz programma GAL (valori teorici)

4. ANALISI DEI RISULTATI OTTENUTI

I risultati ottenuti sono riportati nelle figure allegate.

La figura 2 mostra il comportamento del programma sistolico raffrontato con quelli di due programmi sequenziali scritti in linguaggio C. Quello indicato con la scritta "C iterativo" è lo stesso programma implementato su GAPP, mentre il "C analitico" rappresenta la soluzione classica dei polinomi considerati (i dati rilevati sperimentalmente si riferiscono alla ricerca della prima radice positiva per polinomi di secondo grado).

La scritta "GAL (I/O__Time incluso)" indica che insieme al tempo di CPU è riportato quello di I/O perchè, a causa della limitatezza della RAM (128 bit) accessibile da ogni PE è necessario utilizzare la memoria centrale dell'host (PC XT IBM) come area di parcheggio per i risultati intermedi. L'andamento delle rette indica chiaramente che il programma per il GAPP scritto in linguaggio GAL è notevolmente più "veloce" del suo omologo sequenziale ma meno di quello, pur sequenziale ma analitico, scritto in C (per inciso il numero dei cicli è dovuto al fatto che il programma GAL è iterativo, e approssimativamente occorrono 14 iterazioni per ottenere un risultato con quattro decimali di precisione, operando con numeri interi a 32 bit o in virgola fissa). Il vantaggio del programma C analitico è da attribuire alle differenti frequenze di clock con le quali operano le due macchine. Questo fatto, unito con la possibilità architetturale della scheda di lavorare fino a 10 MHz, giustifica la curiosità di calcolare come l'algoritmo implementato si comporta con tali frequenze di clock.

Nella figura 3 sono mostrati l'andamento teorico del programma GAL raffrontato a quello, reale, C analitico

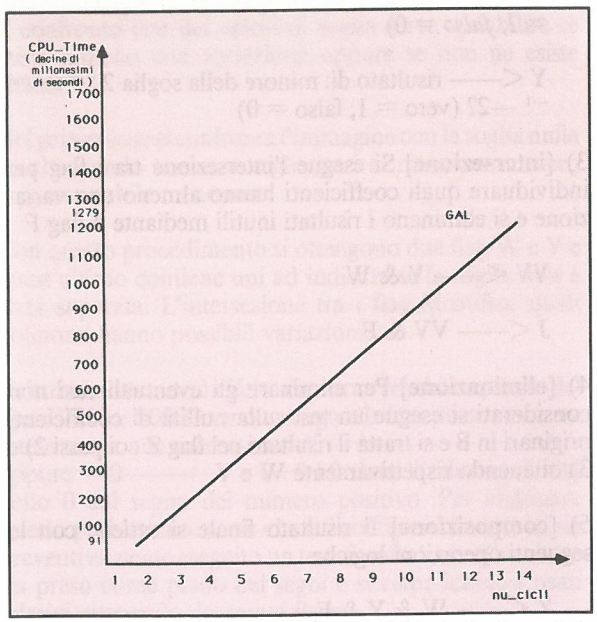

Fig. 4 Ipotesi: 10 MHz programmi GAL (valori teorici)

nell'ipotesi che la frequenza di clock della scheda GAPP sia la stessa di quella dell'host. È evidente il vantaggio dell'algoritmo parallelo su quello sequenziale anche per richieste di precisione molto elevate.

L'andamento dell'algoritmo sistolico "a regime" rappresentato dalla figura 4 nella quale il fattore di guadagno rispetto all'andamento precedente è di 2.1 mentre rispetto ai valori effettivamente misurati è di 333.

Poiché non è stato possibile ricavare sperimentalmente i tempi per l'esecuzione dell'algoritmo su polinomi di gra-

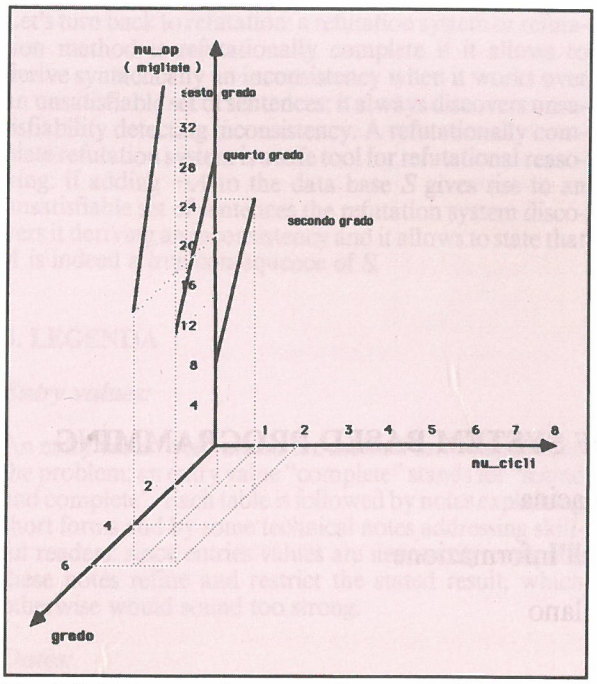

Fig. 5 Relazione tra il numero di operazioni, numero di cicli e il grado del polinomio.

do maggiore di due, la figura 5 indica la relazione esistente tra la precisione voluta, il grado dei polinomi e il numero di operazioni GAPP per ottenere la soluzione desiderata.

Questi valori sono stati utilizzati per mostrare che la migliore tipologia architetturale delle schede di processori per risolvere questo problema è quella di una "macro scheda" composta da più schede base GAPP, rispetto ad una architettura a pipeline, a parità del numero di processori utilizzati.

5. RINGRAZIAMENTI

Desidero ringraziare il Dott. Marini per l'aiuto prestatomi, il Prof. Mauri per i consigli ricevuti e il Prof. Degli Antoni per l'attenzione dimostrata.

6. BIBLIOGRAFIA

Per la documentazione riguardante la scheda GAPP si è fatto uso di manuali della NCR e di compendi della SKYLAB (Milano).

Per una approfondita bibliografia si faccia riferimento a quella riportata in [Zana 87].

[Coll 82] REAL ZEROS OF POLYNOMIALS, C. Collins, R. Loss, Computing Suppl. 4, 1982

[Mari 86] A RAY TRACING SYSTEM FOR IMAGE SYNTHESIS, D. Marini, F. Tampieri, DSI NOTIZIE Num. 0, Febbraio 1986

[Tamp 86] PROBLEMATICHE DI RESTITUZIONE REALISTICA DI SCENE, F. Tampieri, Tesi DSI, 1986

[Zana 87] SINTESI DI IMMAGINE ATTRAVERSO L'IMPLEMENTAZIONE DI ALGORITMI PARALLELI, S. Zanantoni, Tesi DSI, 1987

A THEOREM PROVING REVIEW FOR PROOF SYSTEM BASED PROGRAMMING

Maria Paola Bonacina

Dipartimento di Scienze dell'Informazione

Università di Milano

RIASSUNTO

Questo lavoro è una collezione di tabelle che intendono offrire una rassegna di facile e rapida consultazione sulla dimostrazione automatica di teoremi. La lista dei titoli delle tabelle è la seguente: Primi Precursori, Dimostrazione di Teoremi basata sulla Risoluzione, Ragionare con l'Uguaglianza, A Proposito della Negazione, Ragionare con l'Uguaglianza usando la Riscrittura di Termini, Dimostrazione di Teoremi in Forma a Clausole usando la Riscrittura di Termini, Dimostrazione di Teoremi Basata sulla Riscrittura di Termini.

ABSTRACT

This is a collection of tables intended to give an easy and quick to consult review about automatic theorem proving. The list of tables titles gives the review's contents: Early Forerunners, Resolution Based Theorem Proving, Equality Reasoning, Something About Negation, Term Rewriting Based Equational Reasoning, Term Rewriting Based Clausal Theorem Proving, Term Rewriting Based Theorem Proving.

1. INTRODUZIONE

As the fields of automatic theorem proving and proof systems based programming are growing in several interrelated directions, a review seems to be useful in order to discover historical roots and to compare different approaches and their outcomes.

The present one focuses on two main areas: resolution based theorem proving and rewriting based theorem proving: their counterparts on the programming side are Prolog [13] for the first one, equational languages [5, 6, 14] and rewrite programs [4] for the second one.

This survey is given by a collection of tables organized as *methods* $\times$ *problems* matrices. Problems often appear in more than one table allowing to check how different methods behave on a give problem.

Methods are conceived as black boxes: no explanation about how they work is possible in such a compact survey, but a proper reference is given for each oh them. Fundamental concepts at the core of automatic theorem proving are described in the next section, which mainly addresses unskilled readers.

2. BASIC CONCEPTS

Most of the methods listed in the following tables are refutational. Refutation is the formalization of "reductio ab absurdum": a sentence A is a true consequence of a set of assumptions S if adding A's negation to S gives a contradiction. Formally this reasoning is written: if $S \cup \{\neg A\} \vdash \bot$ then $S \models A$, where $\vdash$ means syntactic derivation, $\models$ means semantic truth and $\bot$ is a short form any contradiction such as $A \wedge \neg A$.

A set of sentences T such that $T \vdash \bot$ is called inconsistent. A set of sentences T such tha there is no interpretation under which every sentence in T is true is called unsatisfiable: there is no way to give a meaning to symbols in T such that every sentence in T is true.

These two concepts, inconsistency and unsatisfiability, should intuitively agree and indeed they do, as stated by the Completeness Theorem of First Order Logic: every set of sentences is unsatisfiable if and only if it is inconsistent.

Let's turn back to refutation: a refutation system or refutation method is refutationally complete if it allows to derive syntactically an inconsistency when it works over an unsatisfiable set of sentences: it always discovers unsatisfiability detecting inconsistency. A refutationally complete refutation system is a safe tool for refutational reasoning: if adding $\neg A$ to the data base S gives rise to an unsatisfiable set of sentences the refutation system discovers it deriving an inconsistency and it allows to state that A is indeed a true consequence of S.

3. LEGENDA

Entry values:

An entry value "yes" means the method can be applied to the problem; an entry value "complete" stands for "sound and complete". Each table is followed by notes explaining short forms and by some technical notes addressing skillful readers: since entries values are necessarily compact, these notes refine and restrict the stated result, which otherwise would sound too strong.

Dates:

These tables should also contribute to an historical survey, so that methods refined or substituted by later ones are listed as well. Dates are taken from [3, 13, 16] and refer to the definition or to the completeness proof of the method.

Warning:

The following tables are mainly oriented to compare the fields of applicability of the listed methods from a completely declarative point of view, ignoring procedural issues such as nontermination and computational complexity: the following completeness results should be read as "complete, unless the procedure loops or aborts".

4. EARLY FORERUNNERS

	Validity in propositional logic	*Unsatisfiability of a set of ground clauses*	*Unsatisfiability of a DNF formula*
Truth Tables (1927) [3]	*complete*		
Multiplication Method (1960) [3]		*complete*	
The Method of Davis and Putnam (1960) [3]		*complete*	
Parwitz Procedure (1960) [3]			*yes*

where

DNF: disjunctive normal form.

5. RESOLUTION BASED THEOREM PROVING

	Unsatisfiability of a set of clauses	*Unsatisfiability of a set of Horn clauses*
Resolution (1965) [3,16]	*complete*	
Set of Support Resolution (1965) [3,16]	*complete*	
Hyperresolution (1965) [3,16]	*complete*	
Unit Resolution (1970) [3,16]		*complete*
Input Resolution (1970) [3,16]		*complete*
Linear Resolution (1968) [3,16]	*complete*	
SLD-Resolution (1974) [13]		*complete*
Model Elimination (1969) [16]	*complete*	
Connection Graph Resolution (1976) [16]	*complete*	

	Unsatisfiability of a DNF formula	*Unsatisfiability of a set of wff*	*T-unsatisfiability of a set of wff*
Connection Method or Generalized Matings (1981) [16]	*yes*		
Non Clausal Resolution (1980) [16]		*yes*	
Theory Resolution (1983) [16]			*yes*

where

DNF: disjunctive normal form

T-unastisfiability: unsatisfiability restricted to the set of interpretations models of a given theory T

wff: well formed formula.

6. EQUALITY REASONING

	Unsatisfiability of a set of clauses with equality	*Unsatisfiability of a set of unit clauses with equality*
Paramodulation (1970) plus Resolution [3,16]	*complete*	
Hyperparamodulation plus Resolution **[3,16]**	*complete*	
Input Paramodulation and Resolution [3,16]		*complete*
Unit Paramodulation and Resolution [3,16]		*complete*
Linear Paramodulation and Resolution [3,16]	*complete*	

where the first method requires only the equality reflexivity axiom $x = x$, while the other methods require also the set of functionally reflexive axioms; in "Input (unit, linear) resolution and paramodulation" the attribute applies to both nouns.

7. SOMETHING ABOUT NEGATION

	Infer $\neg A$ *from P*	*Unsatisfiability of a set of general Horn clauses*
Closed World Assumption (1978) [13]	$P \not\models A$	
Negation as Failure (1978) [13]	$A \in F_P$	
Herbrand rule (1984) [13]	$HMod(comp(P) \cup \{A\}) = \emptyset$	
SLDNF—Resolution (1978) [13]		*sound*

where

P: set of Horn clauses or program

A: any atom

F_P: SLD finite failure set of P

general Horn clause: a clause $A : — B_1,..., B_n$, where A is an atom and $B_1,..., B_n$ are literals, i.e. positive or negative atoms

HMod(A): the set of Herbrand models of A

general program: set of general Horn clauses

comp(P): completion of a general program P SLDNF-resolution plus Negation as Failure.

8. TERM REWRITING BASED EQUATIONAL REASONING

	Simplification	*E-unification*	$e \models \forall_X s[x] = t[x]$	*Unsatisfiability of an extended equational theory*
Demodulation (1967) [16] or rewriting or reduction	*sound*			
Narrowing (1974) [16]		*complete*		
Knuth-Bendix Procedure (1967-1981) [4,11]		*complete*	*complete*	
Unfailing Knuth-Bendix Procedure (1986) [10]		*complete*	*complete*	
S—strategy = UKB + Extened Narrowing + Reflexive Refutation (1986) [10]		*complete*	*complete*	*complete*

where

E: a set of equations

$s[\underline{x}]$: s is a term where variables from the tuple $\underline{x}$ possibly appear

E-unification: $E \models \exists \underline{x}\ s[\underline{x}] = t[\underline{x}]$ which becomes $E \cup \{s[\underline{x}] \neq t[\underline{x}]\} \cup \{\underline{x} = \underline{x}\} \models 1 = 0$ for refutational reasoning

$E \models \forall \underline{x}\ s[\underline{x}] = t[\underline{x}]$ becomes $E \cup \{\hat{s} \neq \hat{t}\} \cup \{\underline{x} = \underline{x}\} \models 1 = 0$ for refutational reasoning

$\hat{s}$: skolemized form of s

Extended equational theory: a finite set of equalities and inequalities with all variables (implicitly) universally quantified.

9. TERM REWRITING BASED CLAUSAL THEOREM PROVING

	Unsatisfiability of a set of clauses	*Unsatisfiability of a set of clauses plus a trs R of domain rules*	*Unsatisfiability of a set of Horn clauses with equality*	$S \models L$
N-strategy (1985) [7]	*complete*			
RN-strategy (1985) [7]		*complete (if R is canonical)*		
RN^+-strategy (1985) [7]		*complete*		
Knuth-Bendix Procedure as in [15] (1985)			*complete*	*complete*

where

trs: term rewriting system

domain rules: non boolean rewrite rules

S: set of Horn clauses with equality

L: positive literal

N-strategy, *RN*-strategy, *RN^+*-strategy require the inclusion of the canonical term rewriting system BA for boolean algebra given in [7] for reduction.

10. TERM REWRITING BASED THEOREM PROVING

	Validity of inductive theorems of an equational theory in its initial model	*Unsatisfiability of a quantifier free formula*	*Unsatisfiability of a set of clauses with equality*	*Unsatisfiability of a set of quantifier free formulas*
Knuth-Bendix Procedure for inductionless induction [4,12] (1982)	*complete*			
Knuth-Bendix Procedure with BA [4] (1983)		*complete*		
En-P-Merge-Para Superposition (1985) [8,9]			*complete*	
Inference System of [2] (1987)				*complete*

where

the Inference system of [2] includes four types of superposition, BN-normalization (BN is a subset of BA), subsumption, formula simplification, term simplification by domain rules, narrowing

the *En-P-Merge-Para*-Superposition method requires BA for reduction and the equational reflexivity axiom $\underline{x} = \underline{x}$ to get refutation as $1 = 0$

the inductionless induction method requires an equational theory satisfying the principle of definition [12]

All the above results about term rewriting based theorem proving relies on fairness in critical pair generation (all critical pairs are eventually generated) and rules orientation (all input or generated equations are eventually transformed into rules); unorientable equations destroy fairness and cause abort except for UKB, *S*-strategy and the Inference system of [2], where the second part of fairness definition is relaxed, *N*-strategy, where all rules are oriented since the right hand side is always 1 or 0, and *RN*-strategy where R is already canonical. It follows that all these methods require a reduction ordering to orient rules, except for UKB, *S*-strategy, *En-P-Merge-Para*-Superposition and the Inference system of [2], where this ordering must also be strong i.e. total on ground terms.

All the listed completion procedures stand to the original one as all the restrictions to resolution stand to general resolution. The main motivation for such a growth of completion procedures is reducing the number of attempted superpositions in order to reduce looping and failing situations and to drive more directly the procedure towards the goal: it is an attempt to embed some control into the inference system itself, since the original Knuth-Bendix Procedure does not allow any control on the completion process.

11. ACKNOWLEDGEMENTS

I would like to thank Giovanni Degli Antoni for useful suggestions and remarks.

12. REFERENCES

[1] L. Bachmair, N. Dershowitz, J. Hsiang, ORDERINGS FOR EQUATIONAL PROOFS In Proceedings, 1st IEEE Symp. on Logic in Computer Science, 346-357, 1986

[2] L. Bachmair, N. Dershowitz, INFERENCE RULES FOR REWRITE BASED FIRST ORDER THEOREM PROVING In Proceedings 2nd IEEE Symp. on Logic Computer Science, 1987

[3] C.L. Chang, R.C. Lee, SYMBOLIC LOGIC AND MECHANICAL THEOREM PROVING, Academic Press, New York, 1983

[4] N. Dershowitz, APPLICATION OF THE KNUTH-BENDIX COMPLETION PROCEDURE, The Aerospace Corporation, Report No. ATR-83(8478)-2, 1983

[5] K. Futatsugi, J.A. Goguen, J.P. Jouannaud, J. Meseguer, PRINCIPLES OF OBJ2 In Proceedings, 12th ACM Symposium on Principles of Programming Languages, 52-66, 1985

[6] J.A. Goguen, J. Meseguer, EQLOG: EQUALITY, TYPES AND GENERIC MODULES FOR LOGIC PROGRAMMING, In D. De Groot, G, Lindstrom eds., Logic Programming. Functions, Relations and Equations, Prentice Hall, 1986

[7] J. Hsiang, RETATIONAL THEOREM PROVING USING TERM REWRITING SYSTEMS, Artificial Intelligence 25, 255-300, 1985

[8] J. Hsiang, REWRITE METHOD FOR THEOREM PROVING IN FIRST ORDER THEORIES WITH EQUALITY, J. Symbolic Computation, 3, 133-151, 1987

[9] J.Hsiang, TWO RESULTS IN TERMS REWRITING THEOREM PROVING, In Proceedings 1st RTA Conference, ed. J.P. Jouannaud, LNCS 202, 301-324, May 1985

[10] J. Hsiang, M. Rusinowitch, ON WORD PROBLEMS IN EQUATIONAL THEORIES, In Proceedings ICALP, 1987

[11] G. Huet, A COMPLETE PROOF OF CORRECTNESS OF KNUTH-BENDIX COMPLETION ALGORITHM, J. Computer and System Sciences, 23, 11-21, 1981

[12] G. Huet, J.M. Hullot, PROOFS BY INDUCTION IN EQUATIONAL THEORIES WITH CONSTRUCTORS, J. Computer and System Sciences, 25, 230-266, 1982

[13] J.W. Lloyd, FOUNDATIONS OF LOGIC PROGRAMMING, Sprnger Verlag, New York, 1984

[14] M. O'Donnell, EQUATIONA LOGIC AS A PROGRAMMING LANGUAGE, MIT Press Cambridge MA, 1985

[15] E. Paul, ON SOLVING THE EQUALITY PROBLEM IN THEORIES DEFINED BY HORN CLAUSES, In Proceedings EUROCAL 85, LNCS 204, Springer Verlag, 1985

[16] M.E. Sitckel, AN INTRODUCTION TO AUTOMATED DEDUCTION, In Fundamentals of Artificial Intelligence, eds. W. Bibel and Ph. Jorrand, LNCS 232, Springer Verlag, 1986

QUADTREE PER LA MEMORIZZAZIONE DEI DOCUMENTI IN FORMATO IMMAGINE SU ELABORATORI PARALLELI

Cinzia Anedda, Leonardo Felician

Istituto di Matematica, Informatica e Sistemistica

Università degli Studi di Udine

RIASSUNTO

In questo lavoro si illustrano i problemi concernenti la memorizzazione di testi e documenti di qualunque tipo nell'ambito aziendale e viene presentato un algoritmo che utilizza una struttura dati gerarchica, il quadtree, come alternativa ai metodi attualmente impiengati per l'archiviazione dei documenti.
Si esamina infine la possibilità di rendere operativo l'algoritmo proposto su un sistema di calcolo largamente pararallelo. Una descrizione delle architetture parallele esistenti precede alcune considerazioni riguardanti la suddivisione dell'algoritmo in "task" che sfrutti al meglio il parallelismo implicito nel codice, allo scopo di rendere minimo il tempo di esecuzione. del programma.

ABSTRACT

In this work problems concerning texts and documents storage are illustred. An algorithm employing a hierachical data structure - the quadtree - is presented as an alternative to traditional methods.
Finally the possibility to implement the proposed algorithm on parallel computers is investigated.

1. INTRODUZIONE

Il problema della acquisizione, catalogazione e distribuzione delle informazioni trova riflessi sempre più marcati sul modo di prendere le decisioni relative alla vita aziendale e in definitiva sulla competitività di un'impresa.

Negli uffici tradizionali i documenti prodotti con un word processor sono costituiti da informazioni codificate e memorizzati su supporto magnetico. Viceversa i documenti-immagine - che includono annotazioni manoscritte, disegni, tabelle, fotografie e anche dati alfanumerici - e tutti i documenti provenienti dall'ambiente esterno, vengono eventualmente fotocopiati, ma comunque restano memorizzati su un supporto cartaceo.

Non è difficile immaginare il costo, in termini di occupazione di spazio, di tempo di accesso alle informazioni, e di gestione che una soluzione di questo tipo comporta.

Una risposta a problemi concernenti l'archiviazione di documenti in formato immagine può essere fornita dalla tecnologia ottica [2]: un lettore ottico, lo scanner, legge il documento originale e trasforma il testo, le figure e ogni simbolo ivi contenuto in segnali digitali, cioè in una sequenza di bit i cui valori, 0 o 1, indicano se un determinato punto del foglio è, rispettivamente, bianco o nero.

La stringa binaria prodotta dallo scanner viene successivamente compressa con un codice di tipo Huffman modificato [7], e infine classificata e archiviata - su disco magnetico oppure ottico - secondo diverse modalità di ricerca.

Sebbene si ottenga un notevole risparmio di spazio in fase di memorizzazione, e una diminuzione dei tempi di trasmissione, che deriva dal fatto di trasmettere la stringa binaria compressa in luogo dell'originale, un documento compresso per mezzo di un codice Huffman è difficilmente trattabile.

Non è possibile effettuare la trasmissione progressiva che consiste nel trasferire la codifica del documento con un

ordine particolare in modo da ottenere una sequenza di immagini sempre più particolareggiate ciascuna delle quali è un'approssimazione di quella originale.

Infine, nel caso si desideri fissare l'attenzione soltanto su una parte del documento, si devono comunque trasmettere per intero le informazioni ad esso associate, poiché i singoli bit della stringa binaria prodotta dallo scanner non contengono informazioni sulla posizione dei corrispondenti punti del foglio.

Per ovviare agli inconvenienti descritti si può adottare un metodo che comprime il documento digitalizzato dallo scanner in modo sostanzialmente diverso, e che unisce alla compressione i vantaggi derivanti dalla trasmissione progressiva e dalla possibilità di visualizzare soltanto alcuni particolari del documento (ad esempio un grafico o una tabella). A tale scopo viene utilizzata una struttura dati gerarchica, il quadtree, che costituisce una generalizzazione degli alberi binari in più dimensioni, ed è considerato un'alternativa alla rappresentazione di un'immagine mediante matrice binaria che si pone l'obiettivo di diminuire l'occupazione di spazio di memoria aggregando zone uniformi della figura.

Da un lato quindi si pongono i metodi di compressione che operano in una dimensione, applicando un algoritmo di tipo Huffman modificato alla rappresentazione riga per riga dell'immagine, mentre dall'altro si collocano le tecniche che operano simultaneamente in due dimensioni e che comprimono il documento grazie alla codifica, mediante quadtree, di aree uniformi della figura.

2. UNA STRUTTURA DATI BIDIMENSIONALE: IL QUADTREE

I quadtree furono introdotti nel 1974 da Finkel e Bentley [4] nell'intento di definire una struttura dati adatta a soddisfare interrogazioni su insiemi di punti dello spazio multidimensionale. Dato un insieme $S=\{P_1,...,P_n\}$ costituito da n punti nello spazio d-dimensionale, ogni punto P_i è una d-upla $(p_1,...,p_d)$ in cui p_j è la j-esima coordinata di P_i. Un quadtree per S si costituisce prendendo una dei punti P_i come radice del quadtree; P_i possiede d coordinate per cui il confronto con un altro elemento dell'insieme porta a 2^d possibili alternative. P_i divide quindi lo spazio in 2^d quadranti e l'insieme S dei punti in 2^d sottoinsiemi che diventeranno i figli di P_i nell'albero.

Il procedimento è ripetuto finché ogni quadrante contiene al più un punto dell'insieme di partenza. Questi nodi finali saranno le foglie del quadtree.

Nello spazio bidimensionale un quadtree è un albero in cui ogni nodo è foglia oppure ha quattro figli che rappresentano i quattro quadranti in cui viene suddiviso l'universo dei dati.

Per poter rappresentare una figura bidimensionale, formata da infiniti punti, si deve innanzitutto rendere finito il numero di punti di cui è composta. Da ciò deriva che un punto non è più un concetto primitivo della geometria, ma ha a sua volta delle dimensioni. Questa nuova entità è detta pixel (da picture element) ed è l'elemento più piccolo visualizzabile sullo schermo di un terminale o su un foglio di carta. Il massimo numero di punti distinti di cui è costituito il lato di un'immagine è la misura della risoluzione dell'immagine stessa.

Le immagini possono essere generate assegnando l'intensità luminosa e il colore ai pixel che la compongono. Essi vengono memorizzati in un vettore detto "frame buffer" che contiene, quindi, una rappresentazione interna dell'immagine. Se si considerano soltanto immagini piane, si può pensare al frame buffer come ad una matrice rettangolare e, sotto questa ipotesi, si denota con pixel sia un punto dell'immagine sia un elemento della matrice ad essa associata.

Per rappresentare un'immagine mediante quadtree la si suddivide progressivamente in quattro parti. Ogni nodo del quadtree è una foglia oppure ha quattro figli che descrivono i quattro quadranti ottenuti suddividendo il quadrante identificato dal nodo padre. La radice dell'albero identifica l'intera immagine e i nodi ai livelli inferiori rappresentano aree sempre più piccole della figura. Le foglie dell'albero corrispondono alle zone uniformi che non richiedono, quindi, ulteriori suddivisioni.

In questo modo, le immagini costituite da vaste zone e colorazione uniforme possono essere altamente compresse. Inoltre, in applicazioni in cui il livello di risoluzione richiesto varia, la trasmissione progressiva dell'immagine è realizzata inviando l'albero in ordine top-down in modo che vengano ricevute innanzitutto aree uniformi dell'immagine, che poi sono completate con particolari sempre più precisi man mano che i livelli inferiori dell'albero sono raggiunti [8]. L'insieme dei nodi presenti su ogni livello p del quadtree è, infatti, una rappresentazione dell'immagine a livello di risoluzione p, formata da blocchi a colorazione uniforme di dimensione 2^P*P^P. Da essa, scendendo di un livello nell'albero, si passa ad una rappresentazione a livello p-1 suddividendo i blocchi in quadranti. Le immagini quindi divono essere quadrate con dimensione pari a 2^n*2^n. Un singolo pixel è identificato da in blocco di dimensione 2^0*2^0, e si trova, quindi, al livello di risoluzione uguale a zero.

È possibile inoltre visualizzare soltanto una parte dell'immagine specificando i confini della regione interessata. Non si dimentichi, infatti, che i quadtree sono stati introdotti come struttura per la ricerca di sottoinsiemi dello spazio multidimensionale.

3. QUADTREE PER MEMORIZZARE I DOCUMENTI

Rivolgendo l'attenzione alle applicazioni di Automazione del Lavoro d'Ufficio (Office Automation) le immagini di cui ci si occupa sono documenti composti da un testo (dattiloscritto o manoscritto) inframezzato da disegni o

grafici, ma in ogni caso le aree a colorazione uniforme non sono di dimensioni molto grandi.

Tenendo presente questa realtà applicativa, l'obiettivo di questo lavoro è analizzare una forma di quadtree che può venir impiegata nella compressione di immagini particolari come i documenti.

A questo proposito, si prendono in considerazione immagini binarie, in cui i pixel possono assumere soltanto due valori, "bianco" o "nero" o - rispettivamente - 0 o 1. La stringa binaria prodotta dallo scanner viene memorizzata in una matrice quadrata che viene progressivamente suddivisa in quattro quadranti di uguali dimensioni. Questi sono scomposti via via in sottoquadrati fino ad arrivare a blocchi (possibilmente singoli pixel) costituiti interamente da zeri o uni.

Il risultato di questo procedimento è un albero in cui la radice corrisponde all'intera matrice, mentre ogni nodo figlio rappresenta uno dei quattto quadranti, chiamati NO, NE, SO, SE, secondo la terminologia geografica, oppure numerati rispettivamente da 0 a 3 (fig. 1) [12, 10]. Le foglie dell'albero corrispondono ai blocchi per i quali non è più necessaria alcuna suddivisione. Un nodo foglia è detto BLACK o WHITE se il blocco che rappresenta è costituito interamente da uni o zeri. I nodi non terminali sono indicati con GRAY.

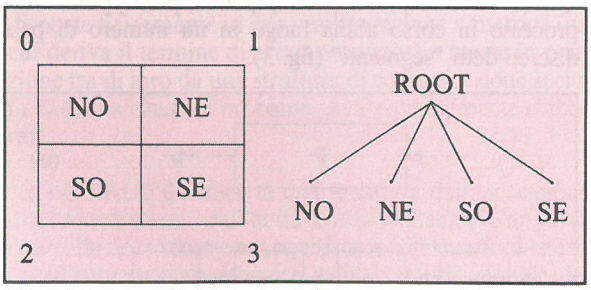

Fig. 1 Divisione dello spazio bidimensionale in quadranti e relativa rappresentazione del quadtree.

I primi studi sui quadtree erano rivolti principalmente alla flessibilità della rappresentazione. Furono quindi sviluppate tecniche che utilizzano puntatori dinamici tra i nodi dell'albero. Ogni nodo è un record costituito da cinque campi, quattro dei quali sono puntatori ai figli NO, NE, SO, SE, e il quinto è un descrittore del nodo che fornisce informazioni concernenti il colore, l'omogeneità, etc.. Sebbene questa struttura, a cui ci si riferisce con il termine di quadtree regolare, fornisca una buona compressione delle immagini, una grande quantità di memoria è utilizzata per i puntatori [9].

Una struttura dati più efficiente è costituita dai quadtree lineari. Il quadtree che viene presentato [6] è basato sull'assegnazione di una chiave numerica ad ogni pixel nero dell'immagine, e sull'utilizzo di tali chiavi per identificare blocchi massimali. Vengono quindi memorizzate soltanto le foglie BLACK del quadtree, e ciò si rivela molto conveniente nel caso particolare dei documenti dattiloscritti in cui i pixel neri sono poco numerosi se confrontati con la totalità dei pixel nel foglio.

Il codice di un pixel è una sequenza di cifre in base quattro che racchiude in sé la sua posizione e riflette le successive divisioni in quadranti secondo la convenzione che associa le cifre 0, 1, 2, 3 ai quadranti NO, NE, SO, SE. Poiché un singolo pixel è il risultato di n suddivisioni, per un'immagine di dimensioni pari a 2^n*2^n, sono necessarie n cifre per il suo codice. In questo modo la cifra che occupa la posizione n-k nel numero, con $1 \leq k \leq n$, identifica il quadrante, a cui il pixel appartiene, alla k-esima suddivisione (fig. 2).

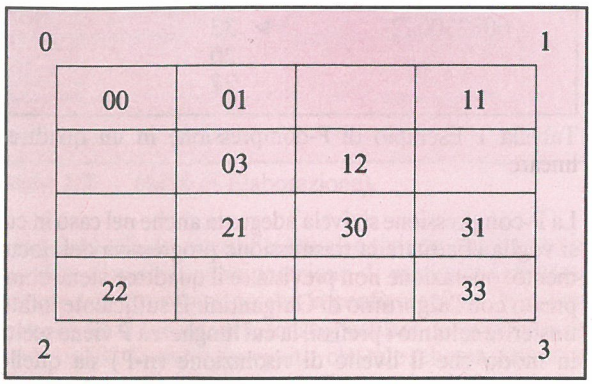

Fig. 2 Esempio di codifica dei pixel.

La procedura di costruzione del quadtree lineare suddivide ricorsivamente la matrice binaria relativa al documento in quadranti e sottoquadranti fino ad arrivare ai singoli pixel che vengono esaminati una ed una sola volta e in modo che è analogo alla visita di un albero in ordine differito. Se si denota la matrice con A, si ha cha A(0,0) è esaminato per primo, seguito da A(0,1), A(1,0), A(1,1), A(0,2), etc., così che gli elementi del quadtree vengono creati in ordine crescente (crf. pseudocodifica).

```
PROCEDURE CONSTRUCT
 BEGIN
  IF "quadrante esaminato contiene un unico pixel"
   THEN
    IF "pixel nero"
     "codifica e memorizza il pixel"
   ELSE
    "suddividi la matrice in quattro quadranti"
    "costruisci il quadtree per il quadrante NO"
    "costruisci il quadtree per il quadrante NE"
    "costruisci il quadtree per il quadrante SO"
    "costruisci il quadtree per il quadrante SE"
 END.
```

A questo punto, anziché comprimere il quadtree con il metodo suggerito da Gargantini [6] si preferisce utilizzare la tecnica denominata P-compressione [1]. Ogni elemento del quadtree si può pensare costituito da un prefisso di P cifre, con $1 \leq P < n$, e un suffisso di (n-P) cifre. Poiché il prefisso si ripete sempre uguale per tutti i pixel neri appartenenti allo stesso blocco individuato dalle prime P suddivisioni - che risulta quindi di dimensione $2^{n-P}*2^{n-P}$ - si ottiene un n otevole risparmio di spazio in memoria

secondaria memorizzando ogni prefisso diverso una sola volta e facendo seguire ciascuno di essi dal numero di elementi del quadtree aventi quel prefisso e dai suffissi corrispondenti (tab. 1)

QUADTREE LINEARE	QUADTREE COMPRESSO
003230023	0032300
003230031	5
003230033	23
003230020	31
003230022	33
	20
	22

Tabella 1 Esempio di P-compressione in un quadtree lineare

La P-compressione si rivela adeguata anche nel caso in cui si voglia effettuare la trasmissione progressiva del documento, operazione non prevista se il quadtree viene compresso con l'algortimo di Gargantini. È sufficiente infatti trasferire soltanto i prefissi, la cui lunghezza P viene scelta in modo che il livello di risoluzione (n-P) sia quello desiderato.

Sebbene l'utilizzo dei quadtree unitamente alla P-compressione fornisca risultati interessanti per quanto riguarda l'occupazione di memoria dei documenti, il tempo di esecuzione non è a vantaggio dell'algoritmo, poiché la procedura di costruzione del quadtree lineare è richiamata un numero di volte pari al numero di nodi in un quadtree completo relativo all'immagine, ed è quindi elevato.

Si ottiene un miglioramento in termini di tempo di esecuzione se il programma viene fatto operare su un calcolatore appartenente alla categoria dei cosiddetti "supercomputer": in tal caso l'algoritmo presentato in questo lavoro potrebbe competere con il metodo di archiviazione dei documenti attualmente usato (che utilizza la scanner), non solo per gli aspetti riguardanti l'occupazione di memoria secondaria, ma anche per quelli concernenti la rapidità con cui le operazioni vengono ultimate.

I problemi inerenti alla disponibilità di grandi potenze di calcolo vengono affrontati muovendosi contemporaneamente in due direzioni: per mezzo della tecnologia si realizzano circuiti sempre più veloci, mentre nell'area sistemistica si punta su architetture largamente parallele. Nel primo caso si può arrivare ad ottenere calcolatori con una velocità di elaborazione di venti volte superiore a quella degli attuali sistemi di calcolo.

Nel secondo caso, invece di un solo processore - spesso progettato fino a raggiungere i limiti della attuali tecnologie - capace di elaborare quantità di dati, si utilizzano tanti processori ciascuno dei quali esegue soltanto una porzione ("task") del programma da elaborare. Naturalmente, in pratica, le cose non sono così semplici e il problema di suddivire un programma in vari "task" in maniera realmente efficiente è attualmente oggetto di studio da parte di molti ricercatori.

4. CENNI DI ELABORAZIONE PARALLELA

Con il termine "parallelismo" si indica il fatto che vengono eseguite in modo sincrono più operazioni dello stesso tipo per volta. Con riferimento allo schema proposto dal Flynn [5], i sistemi possono essere classificati in base a due elementi fondamentali: il numero di istruzioni eseguite simultaneamente dal calcolatore e il numero di operandi su cui esso opera in parallelo.

La soluzione classica, che caratterizza tuttora la grande maggioranza delle macchine, è costituita da un sistema in cui un unico processore (cioè un'unità di controllo e un'unità aritmetico-logica, ALU) esegue le istruzioni serialmente, estraendo un dato per volta dalla memoria.

Accanto ad essa si pongono sia architetture monoprocessore con numerose unità funzionali (tipicamente ALU) sia sistemi multiprocessore e multicalcolatore. Alla prima categoria appartengono quei sistemi il cui processore è costituito, oltre che dall'unità di controllo, da un insieme di ALU, disposte in serie, ognuna delle quali esegue un'istruzione diversa su uno stesso flusso di dati [Fil86]. Questa struttura, nota come pipeline, presuppone che il processo in corso abbia luogo in un numero di passi discreti detti "segmenti" (fig. 3).

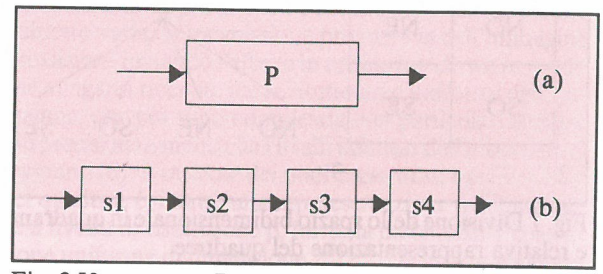

Fig. 3 Un processo P non suddiviso in segmenti (a), e in una forma adatta per essere elaborato su un sistema (b) [13].

Ogni unità opera su dati passati dalla precedente e - al termine della propria elaborazione - passa i risultati alla successiva mettendosi nuovamente in ricezione di dati in ingresso e così via. In ongni istante, a regime, sono in elaborazione tante informazioni distinte e consecutive quante sono le unità, ognuna in una diversa fase dell'elaborazione stessa. I collegamenti tra le unità saranno in generale bufferizzati a seconda delle necessità. I vantaggi derivano dal fatto che non è necessario aspettare che un dato sia completamente trattato prima di passare al successivo e pertanto la struttura è particolarmente utile nel caso di programmi con elevata ripetitività, che esegueno cioè la stessa sequenza di operazioni un numero elevato di volte. L'adozione di una architettura pipeline produce un miglioramento, per quanto riguarda la velocità di esecuzione dei programmi, pari al numero di segmenti in cui è suddiviso il processo, assumenro che tutti i segmenti ven-

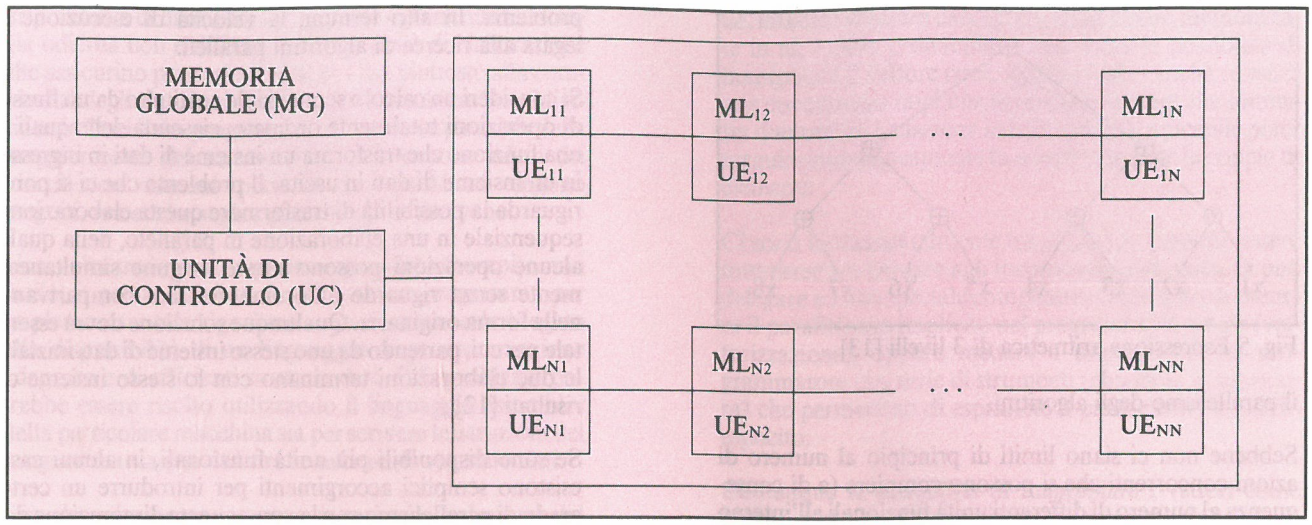

Fig. 4 Schema di un "array processor" (da [12]) (ML = Memoria Locale; UE = Unità di Elaborazione).

gano eseguiti in tempi circa uguali [13].

Sono monoprocessore anche quei sistemi in cui l'unità centrale è costituita da una molteplicità di ALU le quali, però a differenza di quelle presenti in un'architettura pipeline, eseguono la medesima istruzione contemporaneamente su operandi diversi, generalmente appartenenti ad una stessa struttura dati. Queste unità sono spesso disposte fisicamente in una configurazione a matrice (da cui deriva il termine di array processor) e messe in relazione tra di loro da una struttura di comunicazione scelta in base alla classe di problemi più frequentemente elaborati.

Ad ogni ALU è associata una memoria dati, accessibile direttamente solo dall'unità corrispondente. L'unità di controllo del processore acquisisce le informazioni relative all'istruzione da eseguire dalla memoria globale del sistema e le trasmette, insieme agli indirizzi degli operandi, a tutte le ALU, le quali eseguaono l'istruzione contemporaneamente, estraendo gli operandi richiesti dalla loro memoria locale (fig. 4).

Esistono inoltre elaboratori vettoriali (se così si può tradurre il termine array processor) che eseguono la stessa operazione su tutti i bit che costituiscono una riga o una colonna di una matrice di bit di dimensione N*N. I singoli bit vengono esaminati simultaneamente grazie all'utilizzo di una memoria associativa (cioè indirizzabile per contenuto) la quale permette di diminuire i tempi di accesso alle strutture dati con un conseguente aumento della velocità di esecuzione dei programmi. A questo proposito si noti che, mentre in una struttura pipeline l'aumento di velocità è dovuto alla parziale sovrapposizione nel tempo delle singole operazioni, nel caso degli array processor le istruzioni si sovrappongono completamente durante l'elaborazione e pertanto il tempo totale di esecuzione dipende dal tempo necessario per ultimare una singola istruzione.

Schemi come quello appena descritto sono tipici dei grossi calcolatori scientifici attuali e si rivelano adeguati in problemi quali la manipolazione di matrici e l'elaborazione delle immagini poiché la configurazione della matrice di ALU è in un certo senso speculare del problema da trattare [3]. Si vedrà, inseguito, che tale schema si rivela il più adatto per sfruttare il parallelismo implicito nel programma di archiviazione dei documenti.

Finora si è illustrato come ottenre il parallelismo delle operazioni in architetture monoprocessore e si è visto che il motivo che spinge in tale direzione è l'aumento della velocità di esecuzione dei programmi per mezzo di un'attività concorrente sia tra diverse istruzioni sia all'interno di una singola istruzione.

Si accenna infine alla possibilità di configurare sistemi in cui più programmi (o task), ognuno con un proprio insieme di dati, vengono elaborati indipendentemente da più processori capaci anche di interagire tra di loro per un'eventuale cooperazione. Se i processori accedono direttamente alle risorse del sistema (in particolare alle memorie) e fanno parte della struttura interna di una singola macchina fisica, si parla di architettura multiprocessore vera e propria. In caso contrario di è in presenza di un sistema multicalcolatore costituito da un insieme di macchine, anche fisicamente distribuite, la quali accedono a risorse proprie (memorie locali e unità periferiche) e sono collegate tra di loro da mezzi di telecomunicazione.

Da quanto detto è evidente che la concorrenza dei programmi è generalmente realizzata al prezzo di un aumento di complessità soprattutto a livello hardware. Ci si deve quindi attendere l'esistenza di alcuni problemi che pongono dei limiti allo sviluppo cui attualmente si mira, del parallelismo nei sistemi di calcolo. Esistono infatti numerose variabili da considerare in fase di progettazione, ma in questo contesto si tratteranno soltanto i problemi connessi con la suddivisione dei programmi nel più efficiente dei modi, cioè con la ricerca di un algoritmo parallelo per essi, e si accennerà alla situazione riguardante la disponibilità di linguaggi di programmazione capaci di realizzare

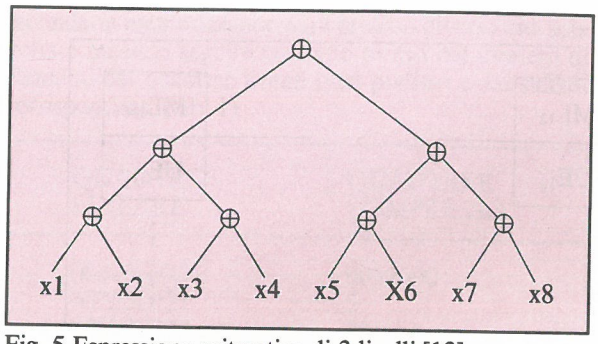

Fig. 5 Espressione aritmetica di 3 livelli [13].

il parallelismo degli algoritmi.

Sebbene non ci siano limiti di principio al numero di azioni concorrenti che si possono compiere (e di conseguenza al numero di differenti unità funzionali all'interno del processore) né al tipo di operazioni ch epossono essere eseguite in parallelo, chiaramente tali limiti si presentano in ogni situazione reale. Infatti non si ricava alcun vantaggio se un gran numero di unità restano inutilizzate perchè ad un certo istante dell'elaborazione non sono disponibili operandi o istruzioni. Comunque la disponibilità di questi ultimi e un numero illimitato di unità funzionali non sono sufficienti a garantire un incremento delle velocità di elaborazione al di là di certi limiti, per un dato algoritmo. Possono esistere infetti vincoli sul numero di operazioni che possono essere eseguite in parallelo, come è illustrato da precedente esempio.

Si consideri la somma di una serie di otto operandi x1,...,x8. A prescindere dal loro ordine, non è possibile valutare l'espressione on meno di tre livelli di un albero binario (fig. 5), anche nel caso in cui non sussistano problemi per quanto riguarda la disponibilità di un congruo numero di sommatori. Inoltre le operazioni ad un dato livello non possono aver inizio finché quelle ai livelli sottostanti non sono state completate [13].

In queste situazioni non rimane altro da fare che scegliere, se possibile, un algoritmo migliore. A prescindere infatti da altre ragioni di insufficienza, il numero di operazioni simultanee non può essere maggiore di quello consentito dallo specifico algoritmo con cui si è risolto il problema. In altri termini, la velocità di esecuzione è legata alla ricerca di algoritmi paralleli.

Si consideri un calcolo sequenziale costituito da un flusso di operazioni totalmente ordinate, ciascuna delle quali è una funzione che trasforma un insieme di dati in ingresso in un insieme di dati in uscita. Il problema che ci si pone riguarda la possibilità di trasformare questa elaborazione sequenziale in una elaborazione in parallelo, nella quale alcune operazioni possono essere eseguite simultaneamente senza riguardo all'ordine con cui comparivano nella forma originaria. Qualunque soluzione dovrà essere tale per cui, partendo da uno stesso insieme di dati iniziali, le due elaborazioni terminano con lo stesso insieme di risultati [12].

Se sono disponibili più unità funzionali, in alcuni casi, esistono semplici accorgimenti per introdurre un certo grado di parallelismo con la conseguente diminuzione del tempo di esecuzione dell'algoritmo. Si può, ad esempio, applicare la proprietà associativa, commutativa o distributiva all'espressione da valutare per ridurre l'altezza dell'albero che la rappresenta (fig. 6) [13].

Più in generale, partendo da una elaborazione sequenziale totalmente ordinata si può ottenere una elaborazione parallela parzialmente ordinata che può essere rappresentata da un grafo, i cui nodi corrispondono alle operazioni e gli archi alle relazioni di precedenza tra le operazioni stesse. L'esistenza di un arco tra l'operazione i e l'operazione j significa che la prima deve essere completata prima che la seconda possa avere inizio.

Possono essere eseguite in parallelo le operazioni cosiddette indipendenti, cioè tali per cui non esiste alcun cammini nel grafo che le comprende entrambe. Inoltre, per assicurare l'unicità dei risultati indipendentemente da come l'elaborazione evolve, è sufficiente che, per ogni coppia di operazioni indipendenti i e j, i risultati della prima non siano dati di ingresso della seconda e, viceversa, i risultati della seconda non siano dati di ingresso della prima; e infine non esistano risultati comuni alle due operazioni [12].

Si può a questo punto concludere affermando che i veri limiti allo sviluppo del parallelismo non sono dovuti alla

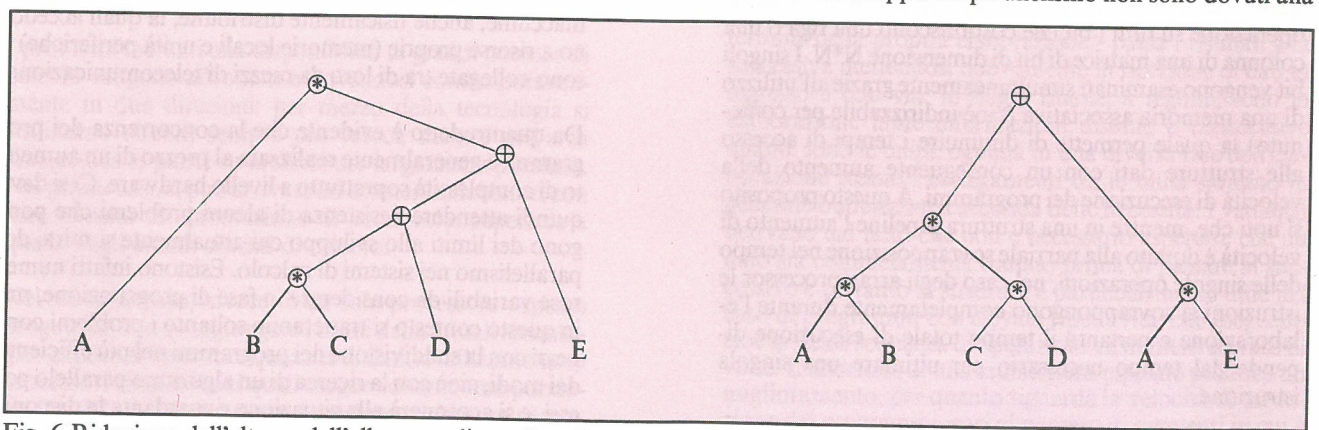

Fig. 6 Riduzione dell'altezza dell'albero mediante l'applicazione della proprietà distributiva all'espressione A (BCD+E) [13].

disponibilità di hardware appropriato - grazie alla tecnologia odierna non è difficile costruire unità di elaborazione che assicurino prestazioni elevate - ma piuttosto alla complessità degli algoritmi che devono rispecchiare la suddivisione in "task" dei programmi. La natura sequenziale della maggior parte dei sistemi di calcolo impone infatti dei vincoli non solo sul modo in cui i problemi vengono eseguiti, ma anche sugli algoritmi che li risolvono.

Una volta trovato un algoritmo parallelo (compito non facile) il problema di come scrivere un programma per implementare tale algoritmo su un sistema reale è legato alla disponibilità di un linguaggio di programmazione adeguato e di efficienti strutture dati. Tale problema potrebbe essere risolto utilizzando il linguaggio assembler della particolare macchina sia per scrivere le istruzioni del programma sia per definire le necessarie strutture dati.

Ciò che veramente interessa è però una soluzione del problema che faccia uso di un liguaggio ad alto livello. Purtroppo, attualmente, non esistono linguaggi di programmazione standard per esprimere il parallelismo delle operazioni ad alto livello e neppure un singolo linguaggio che si riveli adeguato a risolvere il problema della programmazione parallela su un particolare sistema.

In pratica si utilizzano i costrutti di alcuni linguaggi ad alto livello, coma il Fortran, a cui vengono aggiunte chiamate a procedure convenzionali e particolari (che tengono conto cioè di caratteristiche specifiche della macchina) [13] che si rivelano necessarie per introdurre un certo numero di proprietà utili per sfruttare al meglio il parallelismo implicito nei linguaggi di programmazione correntemente in uso. A questo proposito si assiste, da parte dei matematici, a numerosi tentativi che si prefiggono lo scopo di arricchire i linguaggi di programmazione esistenti con costrutti e operazioni particolari che permettano di manipolare i vettori come oggetti unici, definire nuove strutture dati e rendere esplicito il parallelismo.
Per quanto riguarda le strutture dati, uno dei primi ostacoli che deve essere superato (o almeno aggirato) è legato alla rappresnetazione di vettori e matrici, sia come entità con cui operare, sia come oggetti a cui si può accedere in modo opportuno. Il primo aspetto non dipende dalla particolare macchina, anche se alcune proprietà di quest'ultima possono avere un notevole effetto sull'efficienza dell'elaborazione.

Il secondo aspetto è invece strettamente dipendente dalla macchina, in particolare dal modo in cui gli oggetti vengono memorizzati e dall'organizzazione della memoria. Nella maggior parte dei tipici linguaggi di programmazione, infatti le strutture dati, quali i vettori, sono memorizzate in un ordine prestabilito e non esiste la possibilità di manipolare il vettore come oggetto unico anche se esiste una concorrenza implicita nel codice. Si pensi alla somma tra due vettori, situazione in cui sarebbe auspicabile poter eseguire simultaneamente la somma tra tutte le coppie di elementi.

Ci sono fondamentalmente tre strade che possono essere intraprese per ovviare agli inconvenienti descritti. Si può delegare ad uno speciale compilatore il compito di estrarre il parallelismo implicito nel programma sorgente (vettorizzazione), oppure mettere a disposizione del programmatore una serie di strumenti (chiamate a procedura) che permettano di esprimere il parallelismo in modo esplicito.

Comunque la possibilità di manipolare i vettori come entità uniche non è sufficiente a rendere "parallelo" un linguaggio. La terza soluzione consiste pertanto nella difinizione di un nuovo linguaggio di programmazione per l'elaborazione parallela. Inoltre neppure la vettorizzazione risolve tutti i problemi del parallelismo nella manipolazione dei vettori e ciò è a conferma del fatto che soltanto le operazioni indipendenti possono essere eseguite in modo concorrente e inoltre, per garantire l'unicità dei risultati, non ci deve essere alcuna dipendenza dai dati. Si consideri ad esempio il seguente ciclo:

```
FOR I=1 TO 100 DO
  A(I) = A(I—1) + B(I)
```

Esso fornisce risultati diversi a seconda che venga eseguito sequenzialmente o in parallelo. Infatti, poiché l'istruzione ricorsiva, esiste un dipendenza dai dati: A(I) dipende da un valore calcolato all'iterazione precedente, e pertanto no può essere "vettorizzato" dal compilatore [13].

Fortunatamente l'algoritmo di archiviazione dei documenti proposto in questo lavoro non è affetto da tali inconvenienti. La matrice binaria relativa al documento non partecipa ad alcuna operazione aritmetica. Si deve semplicemente testare il valore di ogni pixel e - nel caso in cui quest'ultimo sia nero - eseguire la sua codifica. Si può quindi sperare che in qualche modo possa essere applicata la tecnica della "vettorizzazione".

5. PARALLELISMO DELL'ALGORITMO DI ARCHIVIAZIONE DEI DOCUMENTI

Si esaminano ora tre possibili suddivisioni dell'algoritmo

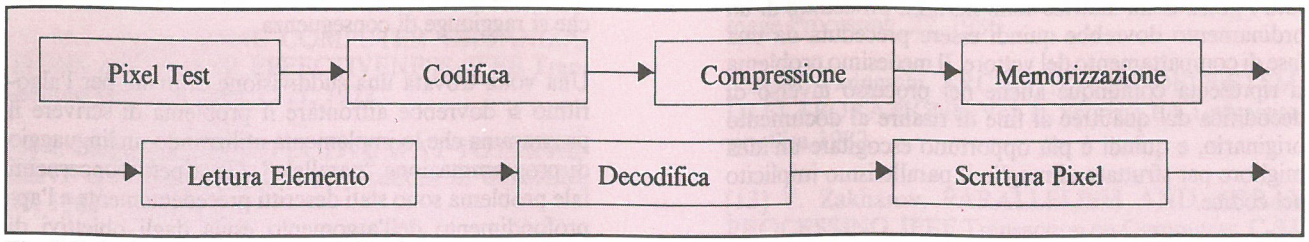

Fig. 7 Soluzione "pipeline" per l'algoritmo.

programma al fine di una sua eventuale esecuzione su un sistema parallelo.

La prima soluzione utilizza un'architettura "pipeline". La matrice binaria viene visitata in modo che i codici posizionali risultino in ordine crescente, e ogni pixel viene testato da una unità di elaborazione. Se il suo valore è 1 allora viene inviato alla successiva unità per la codifica. Il codice che ne risulta viene inseguito compresso e infine memorizzato su disco. Il flusso delle operazioni è illustrato in figura 7.

Naturalmente è l'algoritmo che deve assere adattato al sistema esistente, e non viceversa. Quindi, se le unità di elaborazione sono in numero maggiore di quello dei segmenti del programma, si può ulteriormente scomporre ciascun segmento in alcune fasi di più basso livello.

Nel caso si desideri risalire al documento di partenza si può ragionare in modo analogo al precedente ed effettuare la seguente suddivisione: acquisizione di un elemento del quadtree lineare della memoria secondaria, sua decodifica e infine scrittura del pixel corrispondente nella matrice binaria (fig. 7).

L'adozione di tale soluzione implica dei cambiamenti sostanziali all'algoritmo soprattutto per quanto riguarda la fase di compressione poiché l'architettura "pipeline" non prevede la presenza di memorie locali, associate alle unita, in cui memorizzare i risultati intermedi dell'elaborazione. La procedura di compressione, al contrario, necessità di una struttura dati in memoria centrale poiché la variabile usata come contatore dei suffissi assume il valore esatto soltanto dopo la compressione di un intero blocco. Ad ogni modo, con uno schema come quello descritto, il tempo di esecuzione dell'algoritmo si riduce di un fattore pari al numero di unità di elaborazione utilizzate, ferma restando l'ipotesi che ogni fase dell'esecuzione venga ultimata in un intervallo di tempo uguale agli altri.

La seconda soluzione è più adatta per un array processor, e consiste nell'associare una unità di elaborazione per ogni pixel della matrice binaria. Si ottiene così una matrice di 2^n*2^n unità ciascuna delle quali esegue il test sul pixel corrispondente e, nel caso in cui sia nero, lo codifica. A questo punto gli elementi del quadtree lineare vengono memorizzati simultaneamente in un vettore ("vettorizzazione") e qui ordinati e compressi.

Esiste però un inconveniente: se si assume che ad ogni unità sia associato un elemento del vettore, si verifica la presenza di elementi nulli in tale strutture dati, poiché non tutti i pixell della matrice sono neri. La procedura di un ordinamento dovrebbe quindi essere preceduta da una fase di compattamento del vettore. Il medesimo problema si ripresenta comunque anche nel processo inverso di decodifica del quadtree al fine di risalire al documento originario, e quindi è più opportuno escogitare un'idea migliore per sfruttare al meglio il parallelismo implicito nel codice.

Una volta ottenuta la metrice binaria, risultato del processo di digitalizzazione del documento, l'unità di controllo del processore vettoriale associa le quattro chiamate ricorsive interne alla procedura di costruzione del quadtree lineare ad altrettante unità di elaborazione le quali operano simultaneamente e in maniera del tutto indipendente sui quattro quadranti della matrice al primo livello di suddivisione. Alle unità di elaborazione sono associate altrettante memorie locali, a ciascuna delle quali si accede separatamente. In tal modo, in fase di acquisizione degli operandi (nel caso in esame i singoli pixel) si ottiene un miglioramento del tempo di accesso pari al numero di unità di memoria [13].

Per ogni pixel nero viene creato un elemento del quadtree lineare e al termine, in ognuna della quattro memorie locali è presente una lista che rappresenta il quadtree relativo al quadrante esaminato. Essa viene in seguito compressa e memorizzata su disco.

Il risultato è un insieme di quattro file che si possono concatenare assieme oppure (ed è meglio) mantenere separati per facilitare il processo inverso di acquisizione del quadtree da parte delle quattro unità di elaborazione al fine di risalire al documento originario (fig. 8). Per ogni documento esiste comunque un unico identificatore contenente le informazioni necessarie al reperimento dei quattro quadtree compressi memorizzati du disco.

L'adozione del metodo descritto riduce il tempo di esecuzione dell'algoritmo di un fattore pari al numero di unità di elaborazione ed è facile convincersi che al crescere di quest'ultimo il tempo di esecuzione diminuisce ulteriormente. Quindi, anziché 4 unità di elaborazione, se ne può impiegare 16, valore che equivale al numero di quadranti della matrice al livello di suddivisione uguale a due, oppure 64 (per il livello di suddivisione uguale a tre), 256 e così via, fino al livello di compressione desiderato.

A seconda del valore assunto dal parametro di compressione P, infatti, la dimensione dei quadranti che verranno esaminati dalle unità di elaborazione non deve essere minore del valore $2^{n-P}*2^{n-P}$: in caso contrario, potrebbe succedere che due quadtree distinti contengano elementi appartenenti allo stesso blocco da comprimere e tale situazione comporterebbe la duplicazione di alcune informazioni (il prefisso degli elementi).

Da ciò deriva che il numero massimo di unità di elaborazione che si possono utilizzare per garantire un corretto funzionamento del programma è 2^P*2^P e tale valore corrisponde al fattore di diminuzione del tempo di esecuzione che si raggiunge di conseguenza.

Una volta trovata una suddivisione ottimale per l'algoritmo si dovrebbe affrontare il problema di scrivere il programma che lo implementa utilizzando un linguaggio di programmazione "parallelo". Gli aspetti concernenti tale problema sono stati descritti precedentemente e l'approfondimento dell'argomento esula dagli obiettivi di questo lavoro.

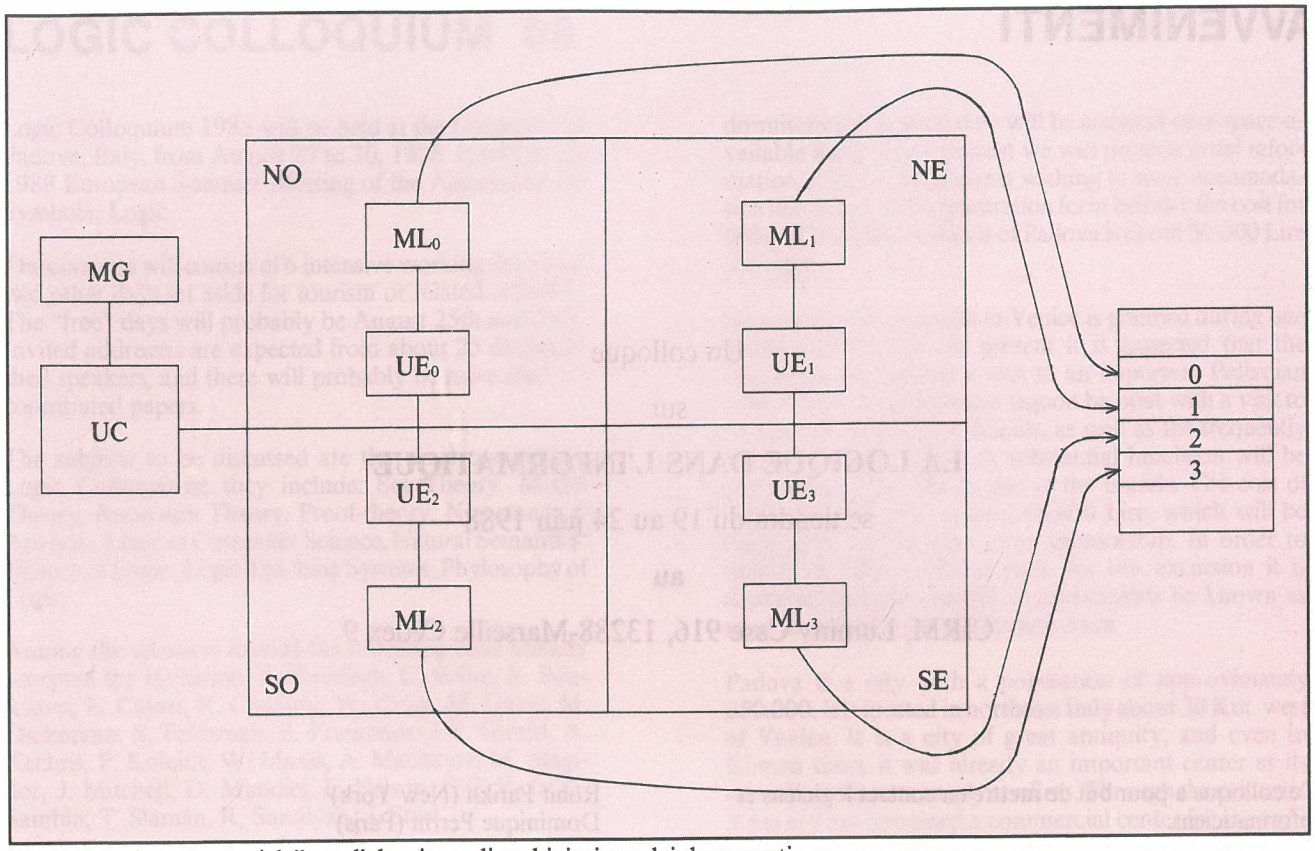

Fig. 8 Soluzione "vettoriale" per l'algoritmo di archiviazione dei documenti.

6. BIBLIOGRAFIA

[1] C. Anedda, APPLICAZIONE DI QUADTREE COMPRESSI ALLA MEMORIZZAZIONE DI GRANDI VOLUMI, Università degli Studi di Udine, Tesi di Laurea, Marzo 1987

[2] L. Felician, VALUTAZIONI HARDWARE E SOFTWARE PER LA GESTIONE DI ARCHIVI IMMAGINI DI GRANDI DIMENSIONI, Rivista di Informatica, 4, 1986

[3] F. Filippazzi, G. Occhini, OLTRE LA MACCHINA DI VON NEUMANN, Quaderni di Informatica, 28, 1986

[4] R.A. Finkel, J.L. Bentley, QUADTREES, A DATA STRUCTURE FOR RETRIEVAL AN COMPOSITE KEYS, Acta Informatica, 4, 1974

[5] M.J. Flynn, SOME COMPUTER ORGANIZATIONS AND THEIR EFFECTIVENESS, IEEE Transactions on Computers, C-21, 1972

[6] I. Gargantini, AN EFFECTIVE WAY TO REPRESENT QUADTREES, Communications of the ACM, 25, 1982

[7] D. Huffman, A METHOD FOR THE CONSTRUCTION OF MINIMUM REDUNDANCY CODES, Proc. IRE, 40, 1952

[8] C. Montani, R. Scopigno, UN SISTEMA INTERATTIVO PER LA TRASMISSIONE PROGRESSIVA DI IMMAGINI, AICA Atti Congresso Nazionale, 1, 1985

[9] H. Samet, REGION REPRESENTATION: QUADTREES FROM BINARY ARRAYS, Computer Vision, Graphics and Image Processing, 13, 1980

[10] I.P. Stewart, QUADTREES: STORAGE AND SCAN CONVERSION, The Computer Jourbal, 29, 1986

[11] M.L.P. Van Lierop, GEOMETRICAL TRANSFORMATIONS AN PICTURES REPRESENTED BY LEAFCODES, Computer Vision, Graphics and Image Processing, 22, 1986

[12] M. Vanneschi, ARCHITETTURA DEI SISTEMI DI ELABORAZIONE, Vol. II, Servizio Ed. Universitario, Pisa 1983

[13] V. Zakharov, PARALLELISM AND ARRAY PROCESSING, IEEE Transaction on Computers, C-33, 1984

AVVENIMENTI

Un colloque

sur

LA LOGIQUE DANS L'INFORMATIQUE

se tiendra du 19 au 24 juin 1988

au

CIRM, Luminy Case 916, 13288-Marseille Cedex 9

Ce colloque a pour but de mettre en contact logiciens et informaticiens.
Il comprendra essentiellement des conférences sur invitation pour laisser sufisamment de temps pour les discussions et l'approfondissement des résultats présentés à l'auditoire.

Les thèmes principaux sont:

algorithmique,
λ - calcul typé
langages de programmation élevés
logiques non-classiques
preuve de correction de programmes.

Conférencies confirmés:

Krzysztof Apt (Amsterdam)
J. Bardsin (Riga)
Andreas Blass (Ann Arbor)
Egon Börger (Pisa)
Ashok Chandra (New York)
Alain Colmerauer (Marseille)
B. Courcelle (Bordeaux)
P. L. Curien (Paris)
Dov Gabbay (London)
L. Esakia (Tbilisi)
David Harel (Rehovot)
Etienne Grandjean (Caen)
Yuri Gurevich (Ann Arbor)
E. Katsov (Moskow)
Jean-Louis Lassez (New York)
Makanin (Moskow)
Daniele Mundici (Milano)
Rohit Parikh (New York)
Dominique Perrin (Paris)
V. Sazanov (Novosibirsk)
D. Scott (Pittsburg)
D. Seese (Berlin)
V. Semyonov (Moscow)
W. Sieg (Pittsburg-München)
Jacques Stern (Paris)
V. Zagarodnyuk (Kiev)

Comité de Programme: E. Börger, (chairman), A. Colmerauer, Y. Gurevich, A. Preller.

Organisateur: A. Preller, Dépt. de Math.-Inf., Faculté des Sciences de Luminy, Université Aix-Marseille II, 70 route Léon-Lachamp, 13288 MARSEILLE CEDEX 9, France.

LOGIC COLLOQUIUM '88

Logic Colloquium 1988 will be held at the University of Padova, Italy, from August 23 to 30, 1988. It will be the 1988 European Summer Meeting of the Association for Symbolic Logic.

The congress will consist of 6 intensive working days with two other days set aside for tourism or related activities. The "free" days will probably be August 25th and 28th. Invited addresses are expected from about 25 distinguished speakers, and there will probably be more than 75 contributed papers.

The subjects to be discussed are the usual ones of the Logic Colloquium; they include: Set Theory, Model Theory, Recursion Theory, Proof-theory, Non-standard Analysis, Logic in Computer Science, Natural Semantics. History of Logic, Logic Teaching Systems, Phylosophy of Logic.

Among the speakers invited the following have already accepted the invitation: A. Baudisch, C. Böhm, E. Bouscaren, E. Casari, R, Chuaqui, W. Craig, M. Davis, M. Dickmann, S. Feferman, S. Friedman, J.Y. Girard, A. Kechris, P. Kolaitis, W. Maass, A. Macintyre, M. Magidor, J. Mitchell, D. Mundici, E. Nelson, G.C. Rota, G. Sambin, T. Slaman, R, Smullyan, J. Steel.

Contributed papers are invited from all areas of Logic: for them a 20 minutes talk is planned. Abstracts, of at most one page, should be sent by April 1, 1988 to: Prof. Franco Montagna, Dip. Matematica, Via Del Capitano 15, 5100 SIENA, ITALY (Tel. 0577/286244).

Abstracts accepted from members of the ASL may be published in the report of the meeting in the Journal of Symbolic Logic.

Scientific Committee: F. Drake, J. Fenstad, R. Ferro, Y. Gurevich, Y. Moscovakis, D. Mundici, A. Scedrov.

Organizing Committee: C. Bonotto, M. Dalla Chiara, R. Ferro, A. Marcja, F. Montagna, D. Mundici, S. Valentini, A. Zanardo.

There will be a registration fee of $70 ($100 after April 1, '88), or $35 for students or accompanying persons ($50 after April 1, '88). Payments should be by cheques made payable to Logic Colloquium '88, and should be sent to the Logic Colloquium address below.

Accomodations: At present two student dormitories are available for use by the participants, the "Antonianum" and the "Ederle". The former offers full board for the entire period of the Colloquium for 350.000 Lire, while the latter offers only sleeping accomodations for 130.000 Lire for the same period, (As of Nov. 1, 1987, 1000 Lire = $0.80). For participants choosing the second alternative, it is anticipated that meals will be available at the student "mensa". Since the number of places in both dormitories is limited, they will be assigned on a space-available basis. Upon request we will provide hotel information to those participants wishing to have accomodation in hotels (see the registration form below): the cost for bed and breakfast in hotels of Padova is about 50.000 Lire for night.

Excursion: An excursion to Venice is planned during one of the "free" days. At present it is expected that the excursion will include a visit to an important Palladian villa, a tour of the Venetian lagoon by boat with a visit to some of its picturesque islands, as well as the frequently vistited sites of Venice. A substantial luncheon will be taken during the visit to one of the islands. The cost of subscription will be around 60.000 Lire, which will be reduced if we can find some sponsorship. In order to obtain the best available price for this excursion it is desirable that the number of participants be known as soon as possible. Cf. registration form.

Padova is a city with a population of approximately 250.000. It is located in northeast Italy about 30 Km. west of Venice. It is a city of great antiquity, and even in Roman times it was already an important center as its ruins from that era amply confirm. Through the centuries it has always remained a commercial center for the agricultural products of its rich surrounding countryside. A large part of Padova's history is bound up with that of Venice to which it is connected by convenient waterways. Its university, which was founded in 1222 (the second oldest in Europe), made it the cultural center of the liberal Venetian Republic. The University of Padova has remained very active through the centuries in fostering the contributions of many renowned scholars. Among these we might mention Galileo, Copernicus, Harvey, Albertus Magnus, Morgagni, Marsilio da Padova, Veronese, Ricci Curbastro. Today the university, with its 50.000 students and 3000 professors, plays a central role in the city whose principla economic activity is now based on service industries. The tourist visiting Padova will find many important artistic works, the fruit of its long cultural history. Moreover, Padova also attracts pilgrims from all corners of the world who come to honor the remains of Saint Anthony.

Getting to Padova. The closest airport to Padova is at Venice, which lies at about an hour's distance by bus. Padova is also easily reachable by train.

The Logic Colloquium address is the following: Logic Colloquium '88 c/o Prof. Ruggero Ferro, Dip. Matematica Pura e Applicata, Via BELZONI 7, 35131 PADOVA, ITALY (Tel. 049/831957).

Honeywell Bull. SOFT.38 M011001 01

CIMINO MICHELE

H B ITALIA-DIREZ.GEN. MARKETING ITALIA

VIA PIRELLI,32
20124MILANO

Ufficio Distribuzione Documentazione Gestione Mailing List
Via del Parlamento, 33 - **20098 BORGOLOMBARDO (MI)**
Trattasi di trasporto di cancelleria-stampati-moduli-documenti
interni-mobili e macchine per ufficio ESONERATI DALL'OBBLIGO
DELLE BOLLE DI ACCOMPAGNAMENTO D.P.R. 6/10/78 n. 627
art. 4 n. 8 e Circolare Ministeriale n. 72 del 23/12/78 paragrafo 10